

AD-A146 030

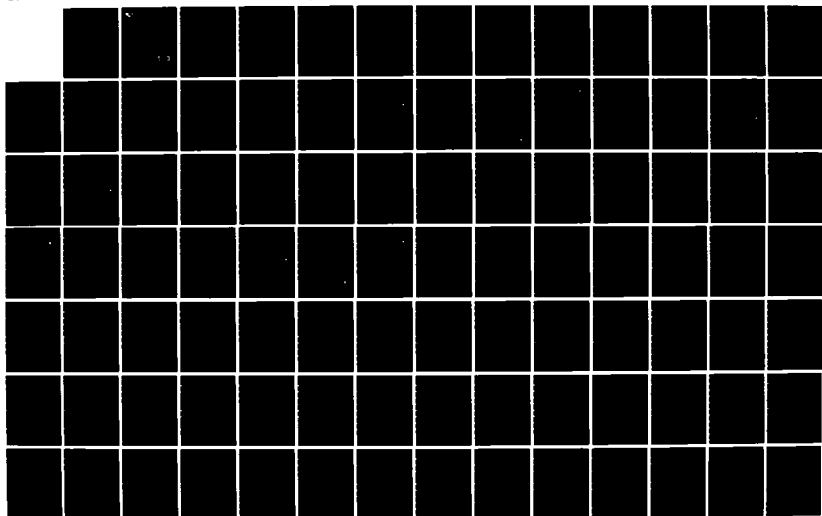
ANALYSIS AND DESIGN METHODOLOGY FOR VLSI COMPUTING
NETWORKS(U) INTEGRATED SYSTEMS INC PALO ALTO CA
H LEV-ARI AUG 84 ISI-46 N00014-83-C-0377

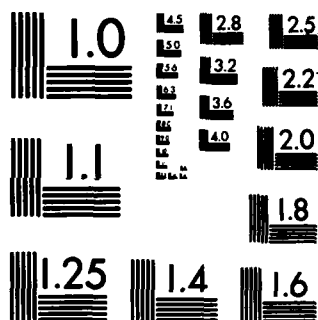
1/2

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



AD-A146 030

ANALYSIS AND DESIGN METHODOLOGY FOR
VLSI COMPUTING NETWORKS

FINAL REPORT

HANOCH LEV-ARI

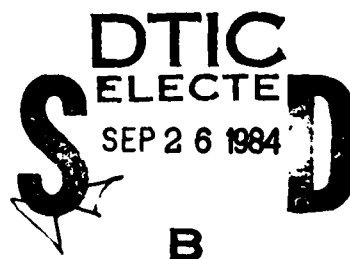
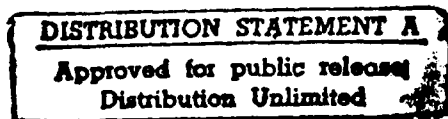
PREPARED FOR:

OFFICE OF NAVAL RESEARCH
800 NORTH QUINCY STREET
ARLINGTON, VIRGINIA 22217

ATTENTION: DR. DAVID W. MIZELL

PREPARED UNDER:

CONTRACT NO. N00014-83-C-0377



ISI REPORT 46 • AUGUST 1984

DTIC FILE COPY

TABLE OF CONTENTS

<u>Section</u>	<u>Title</u>	<u>Page</u>
1	INTRODUCTION	1
2	MODELING PARALLEL ALGORITHMS AND ARCHITECTURES	5
	2.1 Toward a Formal Definition of Algorithms and Architectures	5
	2.2 Modular Computing Networks	8
	2.3 Causality and Executions	16
	2.4 Hierarchical Composition of MCNs	19
	2.5 Comparison of MCNs with Other Network Models	23
	2.5.1 Block-Diagrams and Finite-State Machines	23
	2.5.2 Data-Flow-Graphs and Petri-Nets	25
	2.5.3 High-Level Programming Languages	27
	2.5.4 Summary	28
	2.6 Formal Language Representation of MCNs	28
	2.7 Summary	32
3	STRUCTURAL ANALYSIS OF MCNs	35
	3.1 Numbering of Variables and Processors	35
	3.2 Dimensionality and Order	36
	3.3 Schedules, Delay and Throughput	40
	3.4 Space-Time Diagrams	47
	3.5 Summary	56
4	ITERATIVE AND COMPLETELY REGULAR NETWORKS	57
	4.1 Iterative MCNs and Hardware Architectures	57
	4.2 Completely Regular MCNs	67
	4.2.1 Space-Time Representations in Z^3	68
	4.2.2 Spatial Projection of MCNs in Z^3	70
	4.3 Modular Decomposition of MCN Models	72
	4.3.1 Modular Decomposition of Linear Multivariable Filters	73

5	CLASSIFICATION OF ARCHITECTURES	79
5.1	Topological Equivalence	81
5.2	Architectural Equivalence	85
5.3	Periodicity Analysis and Throughput	85
5.4	Boundary Analysis	88
5.5	Summary	92
6	CLASSIFICATION OF SPACE-TIME REPRESENTATIONS	95
6.1	The Fundamental Space-Time Configurations	95
6.2	Architectures with Local Memory	96
6.3	Boundary Analysis	101
6.3.1	The Configurations LM1, L2, R2	101
6.3.2	The Configurations RM2, R3, H3a, H3b	104
6.3.3	The Configurations HM3a, R4	105
6.3.4	Summary	105
6.4	Interleaving Architectures by Local Memory	105
6.5	Summary	107
7	CONCLUSIONS	109
8	TECHNICAL PUBLICATIONS	111
	REFERENCES	113
	APPENDIX A: PROOF OF THEOREM 2.2 FOR INFINITE MCNs	117
	APPENDIX B: ADMISSIBLE ARCHITECTURES	119
	APPENDIX C: PROOF OF THEOREM 2.3, MINIMAL EXECUTIONS OF FINITE MCNs	121
	APPENDIX D: ELEMENTARY EQUIVALENCE TRANSFORMATIONS	127
	APPENDIX E: ANALYSIS OF MATRIX MULTIPLIERS	129
	APPENDIX F: EQUIVALENCE VIA LINEAR TRANSFORMATIONS	141



Accession For	
THIS GRA&I	<input checked="" type="checkbox"/>
FIG TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
PER LETTER	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

SECTION 1

INTRODUCTION

Several methods for modeling and analysis of parallel algorithms and architectures have been proposed in the recent years. These include recursion-type methods, like recursion equations, z-transform descriptions and 'do-loops' in high-level programming languages, and precedence-graph-type methods like data-flow graphs (marked graphs) and related Petri-net derived models [1], [2]. Most efforts have been recently directed towards developing methodologies for structured parallel algorithms and architectures and, in particular, for systolic-array-like systems [3]-[10]. Some important properties of parallel algorithms have been identified in the process of this research effort. These include executability (the absence of deadlocks) pipelinability, regularity of structure, locality of interconnections, and dimensionality. The research has also demonstrated the feasibility of multirate systolic arrays with different rates of data propagation along different directions in the array.

In this final report we present a new methodology for modeling and analysis of parallel algorithms and architectures. Our methodology provides a unified conceptual framework, which we call modular computing network, that clearly displays the key properties of parallel systems. In particular,

- (1) Executability of algorithms is easily verified.
- (2) Schedules of execution are easily determined. This allows for simple evaluation of throughput rates and execution delays.
- (3) Both synchronous and asynchronous (self-timed) modes of execution can be handled with the same techniques.
- (4) Algorithms are directly mappable into architectures. No elaborate hardware compilation is required.
- (5) The description of a parallel algorithm is independent of its implementation. All possible choices of hardware implementation are evident from the description of a given algorithm. The

equivalence of existing implementations can be readily demonstrated.

- (6) Both regular and irregular algorithms can be modeled. Models of regular algorithms are significantly simpler to analyze, since they inherit the regularity of the underlying problem.

Our methodology is largely based upon the theory of directed graphs and can, therefore, be expressed both informally, in pictorial fashion, and formally, in the language of precedence relations and composition of functions. This duality will, hopefully, help to bridge the gap between the two schools of research in this field. An outline of a formal language representation for modular computing networks is also provided.

The multiplicity of possible hardware implementations for a given computational scheme is efficiently displayed by a space-time representation, a notational tool that has been incorporated into some recent methodologies for modeling, analysis and design of parallel architectures [23-31]. Coordinate transformations of a given space-time representation produce distinct hardware configurations which are equivalent in the sense of being the implementations of the same computational scheme. The problem of mapping a given algorithm into a desired hardware configuration can, therefore, be partly reduced to choosing the appropriate coordinate transformation in space-time. In particular, uniform recurrence relations, which correspond to systolic-array architectures, are described by regular space-time representations. This implies that only linear coordinate transformations are required, and that the entire computational scheme can be described by a small collection of vectors in space-time, the dependence vectors [25,27,28,30]. Consequently, the selection of a desired hardware architecture for a given algorithm reduces to the determination of an appropriate nonsingular matrix with integer entries.

A simple technique for transforming a given 3-dimensional space-time representation into an equivalent canonical form is presented in Sections 5-6. A catalogue of canonical forms is constructed, showing a total of 34 distinct systolic architectures. The task of selecting an appropriate transformation for a given space-time representation reduces, therefore, to the determination of the equivalent canonical form. The important result, which has been overlooked in previous research, is that the canonical equivalent of any given space-time representation is unique. This means

that once a space-time representation has been specified there is no flexibility left in the process of mapping it into systolic-array architectures.

A small fraction of space-time representation does allow some flexibility in selecting the hardware architecture, but only at the cost of inefficient implementation. The well-known example of matrix multiplication, which has four distinct realizations (see [4,5,7,10]) turns out to be one of the few cases where such flexibility is available. A closer examination of the structure of the matrices to be multiplied reveals that each realization is efficient under a different set of structural assumptions (see Section 6.3). Thus, in summary, carefully specified algorithms lead to unique space-time representations which, in turn, lead to essentially unique architectures.

SECTION 2

MODELING PARALLEL ALGORITHMS AND ARCHITECTURES

The concepts of 'algorithm' and 'architecture,' which have been widely used for several decades, still seem to defy a formal definition. Books on computation and algorithms either take these concepts for granted or provide a sketchy definition using such broad terms as 'precise prescription,' 'computing agent,' 'well-understood instructions,' 'finite effort' and so forth. The purpose of this section is to provide a simple formal model for modeling and analysis of (parallel) algorithms and architectures. This model, which we call modular computing network (MCN) exhibits all the properties usually attributed both to algorithms and to hardware architectures. As a first step toward the formal introduction of this model we extract in Section 2.1 the main attributes of algorithms from their characterizations in the literature. This analysis of literature leads to the conclusion that algorithms can only be defined in a hierarchical manner, i.e., as well-formed compositions of simpler algorithms, and that the simplest (non-decomposable algorithms) cannot and need not be defined. The building blocks of the theory of algorithms are characterized in terms of three attributes: Function (what building blocks do), execution time (how long they do it), and complexity (what does it cost to use them). These observations are incorporated into the modular computing network model, as described in Sections 2.2 - 2.6.

2.1 TOWARD A FORMAL DEFINITION OF ALGORITHMS AND ARCHITECTURES

In this section we attempt to extract the main attributes of algorithms and architectures from a randomly chosen sample of 'definitions.' Most characterizations of algorithms are geared to the notion of sequential execution. Nevertheless, we shall see that this underlying assumption is

almost never made explicit. As a result, the attributes of parallel algorithms are, in fact, included in the available characterizations.

As a typical example consider the following definition. 'The term 'algorithm' in mathematics is taken to mean a computational process, carried out according to a precise prescription and leading from given objects, which may be permitted to vary, to a sought-for result' [11]. This definition simply states that an algorithm is a well-defined input-output map and that its domain contains at least one element, and usually more than one. However, the term 'computational process' hints that an algorithm is more than just a well-defined function. Indeed, 'A function is simply a relationship between the members of one set and those of another. An algorithm, on the other hand, is a procedure for evaluating a function' [12].

But how are functions evaluated? We are told that 'this evaluation is to be carried out by some sort of computing agent, which may be human, mechanical, electronic, or whatever' [12]. Thus, the emphasis is on physical realizability (the existence of a 'computing agent') but not on the actual details of the realization. The first axiom of the theory of algorithms is, therefore:

There exist basic functions that are physically realizable.

Further efforts to define physical realizability turn out to be quite futile. This is recognized by Aho, Hopcroft and Ullman who say, 'each instruction of an algorithm must have a 'clear meaning' and must be executable with a 'finite amount of effort.' Now what is clear to one person may not be clear to another, and it is often difficult to prove rigorously that an instruction can be carried out in a finite amount of time' [13]. Physical realizability is a matter of technology: What is non-realizable today may become realizable in a year or two. The theory of algorithms has to assume the existence of realizable basic input-output maps but need not be concerned with the details of their implementation. Therefore, the core of any theory of algorithms is a non-empty collection of undefined objects, which we shall call processors. These are the 'computing agents' mentioned above, and they are assumed to have three attributes:

- (i) Function (an input-output map)
- (ii) Complexity measure
- (iii) Execution time

A processor is assumed to be capable of evaluating the input-output map in the specified execution time. The cost of utilizing the processor is specified by its complexity measure. Notice that the notion of 'effort' mentioned above is a combination of the processor's complexity and its execution time.

It is important to draw a distinction between an algorithm and its description. An algorithm consists of processors (or basic functions), corresponding to all the functions that need to be evaluated. For instance, the computation of $\sin x$ via the first 100 terms of its MacLaurin series requires 100 basic functions, one for each term of the truncated series. The description of the same algorithm in terms of instructions requires only one instruction, which will be repeated 100 times with varying coefficients. Since descriptions of algorithms need to be communicated, stored and implemented, they must be finite, i.e., contain a finite number of instructions. The algorithm itself, on the other hand, may consist of an infinite number of processors, and used to process an infinite number of inputs into an infinite number of outputs. Such are, for instance, most signal processing algorithms: Their inputs and outputs are time-series which may, in principle, be infinitely long. The executability of these algorithms depends upon their capability to compute any specific output with finite time and effort, and to use only a finite number of inputs for this purpose. This observation also sheds a new light on the concept of 'termination,' which is usually overemphasized in definitions of algorithms.

The basic functions comprising an algorithm are interdependent in the sense that the outputs of one processor may serve as inputs to other processors. A complete characterization of an algorithm requires, therefore, to specify both its basic operations and the interconnection between these operations. The same statement applies, of course, to block-diagram representations of hardware, to flow-graphs and, in fact, to any network-type schematic. While algorithms are commonly described in some

formal language, they can also be described in a schematic manner. Conversely, schematic hardware descriptions can be transformed into formal language representations. To emphasize this equivalence we shall introduce the concept of a modular computing network (MCN), which exhibits the common attributes of both algorithms and architectures. Thus, an MCN is a pair

$$M = \{F, G\}$$

where F , the function of the network, is essentially the collection of basic functions discussed above, and G , the architecture of the network, is a directed graph describing the interconnections between basic functions. A detailed definition is provided in Section 2.1.

The concept of modular computing network is hierarchical by nature. Basic functions can be themselves characterized as networks composed of more basic functions. This requires every MCN to have the three fundamental attributes of a basic function: Input-output map, complexity, and execution time. We shall show in the sequel how to uniquely associate such attributes with modular computing networks. The theory of MCNs is, in short, the theory of network composition (deducing the properties of a network from its components) and network decomposition (characterizing the components and structure of a network whose composite properties have been specified).

2.2 MODULAR COMPUTING NETWORKS

A modular computing network (MCN) is a system of interconnected modules. The structural information about the network is conveyed by specifying the interconnections between the modules, most conveniently in the form of a directed graph (Figure 2-1). The functional information about the network is conveyed by characterizing the information transferred between modules and the processing of this information as it passes through the modules.

The structural attributes of an MCN are completely specified by its architecture, which is an ordered quadruple

$$\text{Architecture} = \{S, T, A, P\} \quad (2.2)$$

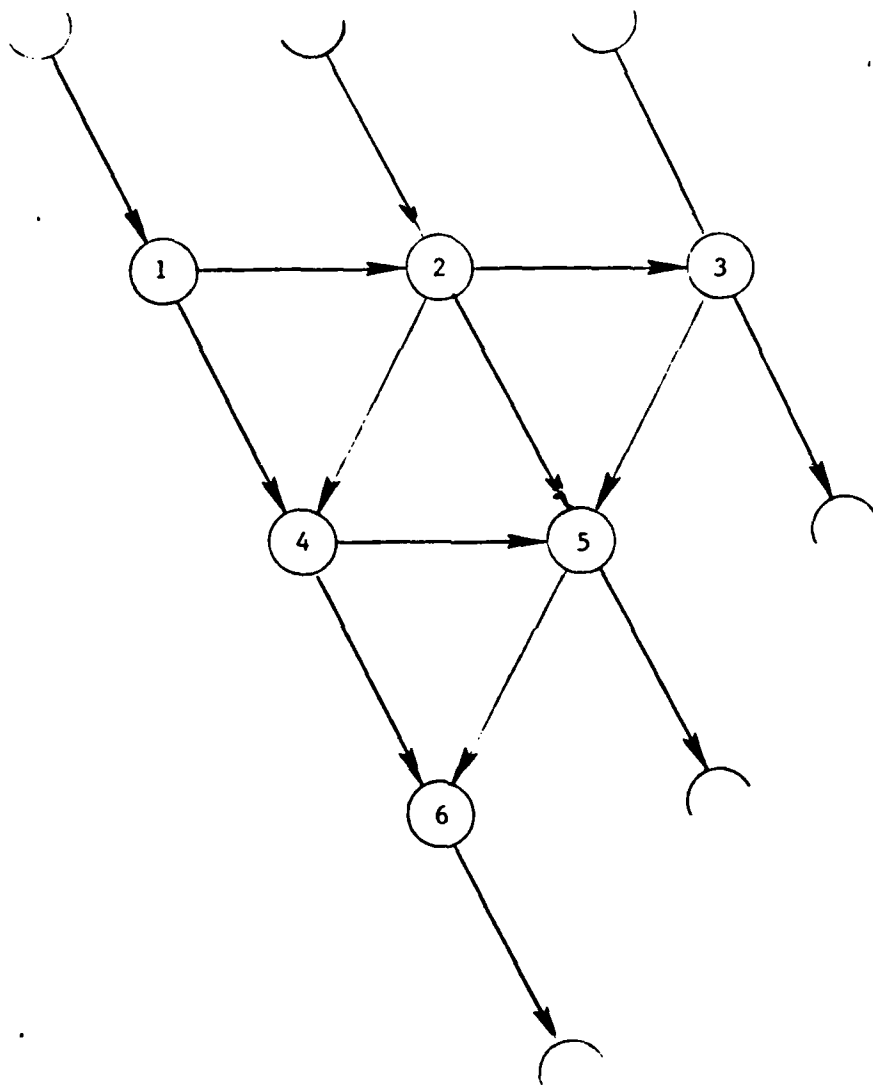


Figure 2-1. The Directed Graph Associated with a Modular Computing Network

where S, T are sets whose elements are called sources and sinks, respectively, and A, P are relations between these sets.

The ancestry relation A specifies the connections of sources to sinks. The elements of A , which are called arcs, are ordered source-sink pairs

$$a \in A \Rightarrow a = (s, t), \quad s \in S, \quad t \in T \quad (2.3)$$

An arc represents a direct transfer of information from source to sink. Two basic assumptions govern this transfer:

- (1) There are no dangling sources. Every source is connected to exactly one sink.
- (2) There are no dangling sinks. Every sink is connected to exactly one source.

These assumptions mean that the three sets S, T, A have an equal number of elements, and that the ancestry relation A establishes a one-to-one correspondence between arcs, sources and sinks, viz.,

$$(s, t) \in A \Leftrightarrow s = A(t) \Leftrightarrow t = A^{-1}(s) \quad (2.4)$$

This one-to-one correspondence will permit us to identify in the sequel each arc with its associated source and sink, and to eliminate almost all sinks and sources from the description of network architectures.

The processing relation P specifies the processing of information extracted from sinks into transformed information, which is re-injected into sources. The elements of P , which are called processors, are ordered pairs of non-empty finite sink-source sequences, viz.,

$$p \in P \Rightarrow p = \{t_1, t_2, \dots, t_m; s_1, s_2, \dots, s_n\} \quad (2.5)$$

$$t_i \in T, \quad s_j \in S, \quad 1 \leq m, n < \infty$$

The input set (t_1, t_2, \dots, t_m) consists of all the sinks from which the processor p extracts information. The transformed information is distributed among the members of the output set (s_1, s_2, \dots, s_n) . The one-to-one correspondence between sources, sinks and arcs allows us to describe processor inputs and outputs in terms of arcs and to almost completely eliminate the notion of sources and sinks. The set of input arcs of a processor p is denoted by $A_i(p)$, and the set of output arcs from the same processor is denoted by $A_o(p)$. Each processor is assumed to have unique inputs and outputs, namely

$$p, q \in P, \quad p \neq q \implies \begin{cases} A_i(p) \cap A_i(q) = \phi \\ A_o(p) \cap A_o(q) = \phi \end{cases} \quad (2.6)$$

Similarly, every collection of processors, $Q \subset P$, has its uniquely defined inputs and outputs, viz.,

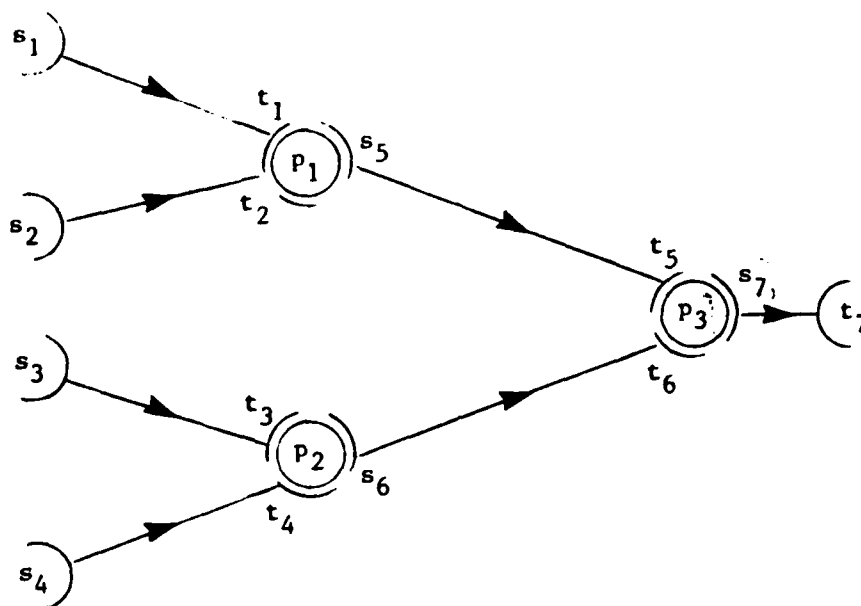
$$A_i(Q) := \bigcup_{p \in Q} A_i(p) - \bigcup_{p \in Q} A_o(p) \quad (2.7a)$$

and

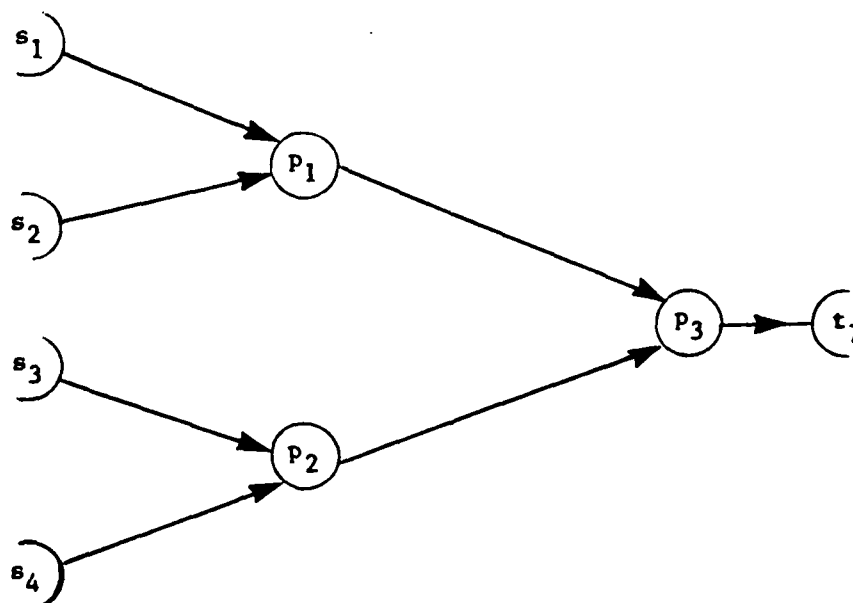
$$A_o(Q) := \bigcup_{p \in Q} A_o(p) - \bigcup_{p \in Q} A_i(p) \quad (2.7b)$$

In other words, the inputs of Q are those inputs of processors in Q that are not connected to outputs of processors in Q . A similar statement holds for outputs of Q . In particular, $A_i(P)$, $A_o(P)$ are the inputs and outputs of the entire network.

Network architectures are most conveniently described by a directed graph that combines together the ancestry relation A and the processing relation P into a single block-diagram-like representation (Figure 2-2a). Sources and sinks are denoted by semi-circles, processors by circles and arcs are, obviously, denoted by arcs. Sources and sinks are paired, and each processor has its inputs and outputs adjacent to itself. An obvious reduction in notation (Figure 2-2b) enhances the comprehensibility of the description. The reduced form is, essentially, a block-diagram representation of the network architecture, and can be interpreted as a directed graph \mathcal{G}



a. Full Form Description



b. Reduced Form Description

Figure 2-2. Equivalent Full Form and Reduced Form Descriptions of Network Architectures

$$\mathcal{G} = \{V, A\} \quad (2.8a)$$

The set of vertices V of this graph is

$$V = \{A_i(P), P, A_o(P)\} \quad (2.8b)$$

where $A_i(P)$ are interpreted as the sources corresponding to the input arcs and $A_o(P)$ are interpreted as the sinks corresponding to the output arcs. The arcs of the directed graph coincide with the original set of arcs A . The interpretation of network architectures as directed graphs puts at our disposal the powerful tools and results of graph theory. Some of these will be used in the sequel to characterize and analyze the structure of modular computing networks.

The functional attributes of an MCN are completely determined by its architecture and by specifying the functional attributes of each processor. Thus, the function of a network is an ordered pair

$$\mathcal{F} = \{X, F\} \quad (2.9)$$

where X, F are sets whose elements are called variables and maps, respectively.

The elements of X are sets (i.e., domains) and 'assigning a value to a variable' amounts to choosing a particular element in the domain corresponding to that variable. There is one variable, x_a , associated with every arc $a \in A$ of the corresponding architecture. Consequently, there is a one-to-one correspondence between variables, sources, sinks and arcs. This correspondence makes it possible to refer to the variables associated with the inputs of a given processor p as the input variables of p and denote them by $X_i(p)$. A similar notation, $X_o(p)$, is used for the variables associated with the outputs of the processor p .

The elements of F are multivariable maps. There is one map, f_p , associated with every processor $p \in P$ of the corresponding architecture. It maps the input variables of this processor into the corresponding output variables, viz.,

$$f_p : X_i(p) \rightarrow X_o(p) \quad (2.10)$$

which means that each of the output variables is a function of the input variables (not necessarily of all the input variables). This establishes a precedence relation between the inputs and outputs of a given processor, viz.,

$$x \rightarrow y \quad (2.11)$$

if $x \in A_i(p)$, $y \in A_o(p)$ and if y is a function of x (and, possibly, of other input variables). The transitive closure of this relation is also a precedence (i.e., a partial order): We shall say that x_1 precedes x_n if there exists a sequence of variables such that

$$x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n$$

in the sense of (2.11). This global precedence will also be denoted by $x_1 \rightarrow x_n$. The ancestry [14] of a variable $x \in X$ is the set of all variables that precede x , viz.,

$$\alpha(x) := \{z; z \in X, z \rightarrow x\} \quad (2.12)$$

These are all the variables that have to be known in order to determine the value of x .

Since the function of a network consisting of a single processor p is

$$\mathcal{F}_p = \{X_i(p) \cup X_o(p), f_p\}$$

there is, essentially, no distinction between the function and the map of p . Thus, the input-output map of a single processor may also be called the function of the processor. The same is not true for a network consisting of several processors: The input-output map of a network is a single multivariable map, relating the outputs of the network to its inputs; the

function of the network, in contradistinction, is the collection of the atomic maps that comprise the network. The analysis problem for computational networks is to determine the network map from its function. The synthesis problem is to design an MCN (i.e., specify its structure and function) that realizes a given multivariable input-output map.

Modular computing networks need not be finite. In fact, most signal processing algorithms correspond to infinite MCNs. However, the concept of finite effort, involved in the evaluation of variables, imposes certain constraints upon infinite networks. First, the number of inputs and outputs of every processor must be finite. This means that the graph \mathcal{G} describing the architecture is locally finite [15]. Next, every variable must be computable with finite effort, so it will be required to have a finite ancestry, viz.,

$$|\alpha(x)| < \infty \quad \text{for all } x \in X \quad (2.13)$$

We shall also assume that the number of connected components of the architecture \mathcal{G} is countable. A modular computing network that satisfies the three assumptions stated above--local finiteness, finite ancestry and countable number of connected components--will be called structurally finite. The following result characterizes the kind of infinity allowed in such networks.

Theorem 2.1

A structurally finite MCN has a countable number of variables and processors. The following three statements are equivalent:

- (1) The number of variables is finite.
- (2) The number of output variables is finite.
- (3) The number of processors is finite.

Proof:

The countability of the variables and processors of a connected network is a direct consequence of local finiteness (see, e.g., [15]). Since each connected component has a countable number of variables and processors, the same is obviously true for a countable number of connected components. Thus the number of variables and processors of a structurally finite MCN must be countable. As a consequence of local finiteness, a finite number of processors implies a finite number of variables and vice versa, so (1) and (3) are equivalent. Clearly (1) implies (2), while (2), via the finite ancestry condition, implies (1).

2.3 CAUSALITY AND EXECUTIONS

The definition of processors in the previous section did not take into account any constraints imposed by hardware implementation considerations. the most important among these constraints is the causality property. It will be henceforth assumed that an output of a processor cannot become available before the inputs of the same processor that precede this output became available. In the beginning all variables are unavailable; the inputs of the network are made available at a given instant, and following that event, all variables of the network gradually become available. This temporally ordered process, which we shall call execution, must be consistent with the precedence relation between variables induced by the directed nature of the architecture \mathcal{G} . A network that possesses an execution in which every variable ultimately becomes available is said to be executable (or 'live' in the terminology of Petri-nets [1]). It is clear that a network containing a cycle cannot be executable since every variable (= arc) on the cycle can never become available. In order to satisfy the causality assumption every variable in the cycle must temporally precede itself (i.e., it must be available before it becomes available (Fig. 2-3)), which is, clearly, impossible. It turns out that every acyclic architecture is executable. To prove this result we shall need to formalize the notion of execution.

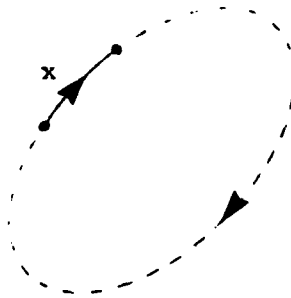


Figure 2-3.

An execution of an MCN is a partitioning of its variables into a sequence of finite disjoint sets, viz.,

$$E = \{S_i; 0 \leq i < \infty, |S_i| < \infty, S_i \cap S_j = \emptyset \text{ for } i \neq j, \bigcup S_i = X\} \quad (2.14a)$$

such that the precedence relation is preserved, viz.,

$$a(S_i) \subset \bigcup_{j=0}^{i-1} S_j \quad i = 0, 1, \dots \quad (2.14b)$$

Here $a(S)$ denotes the ancestry of the set S , defined as the collection of all ancestors of members of S , viz.,

$$a(S) := \bigcup_{x \in S} a(x) \quad (2.15)$$

In simple words, every ancestor of $x \in S_i$ must be contained in one of the sets S_0, S_1, \dots, S_{i-1} , which we shall call levels. Executions can be interpreted as multistep procedures for evaluating all the variables in X . The members of the level S_i are evaluated at the i -th step, and the

condition (2.14b) guarantees the availability of all their ancestors at the right moment. Since the ancestors of the level S_i strictly precede S_i all variables in this set can be evaluated simultaneously giving rise to a parallel execution. If each set S_i contains exactly one variable the execution will be called sequential.

Since each level S_i in an execution is finite, the evaluation of the variables in S_i from the members of the preceding levels requires finite effort. Since each variable belongs to some level S_i , the total effort involved in the evaluation of a single variable from the global inputs is also finite. Thus, the existence of an execution for a given MCN implies that every variable can be evaluated with finite time and hardware. A network that has an execution deserves, therefore, to be called executable.

The preceding discussion implies that executability is a structural property, since only the precedence relation between variables is involved in constructing executions. The following result presents a simple structural test for executability of MCNs.

Theorem 2.2

A structurally finite MCN is executable if, and only if, its architecture is acyclic.

Proof:

If an execution exists, then it can be easily converted into a sequential execution by ordering the variables in each (finite) level S_i in some arbitrary manner. Thus, executability is equivalent to the existence of a sequential execution. By a well-known result in the theory of finite directed graphs, a sequential ordering exists if, and only if, the graph is acyclic. Thus, the theorem holds for finite MCNs. The proof for infinite networks is given in Appendix A.

Corollary 2.2

Executable MCNs always have sequential executions.

The corollary confirms the intuitive notion of executability: Any computation that can be carried out in parallel can also be carried out sequentially. Parallel execution offers, however, an attractive trade-off between hardware and time, which will be discussed in detail in Sec. 3.4.

Theorem 2.2 provides a simple test for executability and, in effect, prevents the construction of non-executable MCNs. Thus, the pitfalls of starvation and deadlocks, well known in the context of Petri-nets [1] are easy to avoid. Notice also that since each variable in an MCN is evaluated exactly once, safeness [1] is guaranteed. This means that inputs to processors do not disappear before they have been used to evaluate the subsequent outputs. Safeness is achieved because once a variable becomes available it stays so forever, and never disappears.

2.4 HIERARCHICAL COMPOSITION OF MCNs

Modular computing networks are, by definition, constructed in a hierarchical manner. A processor p in an MCN can itself be a network, provided it has a well defined input-output map f_p . In this section we analyze the constraints that have to be imposed upon MCN composition in order to guarantee the existence of a well-defined global input-output map.

From the structural point of view a composition is simply a network of networks. The 'processors' of the composite network are MCNs and the arcs represent interconnections between outputs of MCNs to inputs of other MCNs. The architecture of the composition, obtained by regarding each MCN component as a simple 'processor' has to satisfy the constraints of Sec. 2.2. An architecture is called admissible if it satisfies the three following constraints:

- (1) No dangling inputs and outputs
- (2) No cycles
- (3) It is structurally finite

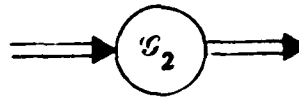
The importance of these constraints lies in the fact that an admissible composition of admissible architectures is itself an admissible architecture (see Appendix B for proof). It is interesting to notice that the admissibility conditions are instrumental also in establishing other important properties of architectures. In particular, an admissible composition of self-timed elements is itself a self-timed element [6], [7].

To establish the hierarchical nature of composition it is only necessary to verify that an admissible composition of processors with a well-defined input-output map also has a well defined input-output map. This will be done by interpreting executions as decompositions of MCNs into elementary parallel and sequential combinations.

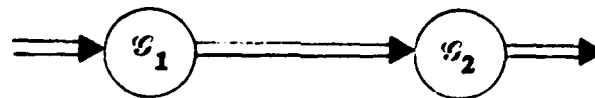
Parallel composition of two architectures, \mathcal{G}_1 and \mathcal{G}_2 , is defined as the union of the two networks without any interconnections between \mathcal{G}_1 and \mathcal{G}_2 (Fig. 2-4a). Sequential composition involves the connection of every output of \mathcal{G}_1 to a corresponding input of \mathcal{G}_2 ; thus the number of outputs of \mathcal{G}_1 must equal the number of inputs of \mathcal{G}_2 (Fig. 4-2b). We shall denote parallel composition by $\mathcal{G}_1 \# \mathcal{G}_2$ and sequential composition by $\mathcal{G}_1 * \mathcal{G}_2$. The parallel composition of a countable number of admissible networks is always admissible. The sequential composition of a sequence of admissible networks is admissible too, i.e.,

$$\mathcal{G}_1 * \mathcal{G}_2 * \dots$$

is admissible because the unilateral nature of the cascade preserves the finite ancestry property, while local-finiteness and countability of components are clearly preserved.



a. Parallel Composition $S_1 \# S_2$



b. Sequential Composition $S_1 * S_2$

Figure 2-4. Fundamental Architecture Compositions

Executions define a rearrangement of MCNs as a sequential composition of subnetworks, each subnetwork being a parallel composition of processors. The MCN of Figure 2-1 can, for instance, be described as

$$(f_1 \# e \# e) * (e \# f_2 \# e) * (f_4 \# e \# f_3) * (e \# f_5 \# e) * (f_6 \# e \# e)$$

where e is an identity input-output map. The importance of this observation lies in the fact that the input-output map of any sequential-parallel composition is well-defined. Consequently, every execution has a well-defined input-output map. This leads to the following result.

Theorem 2.3

Every executable MCN has a unique well-defined input-output map.

Proof:

See Appendix C.

The theorem establishes the utility of the notion of execution. While each execution corresponds to a different ordering of the computations required to evaluate the output variables of an MCN, all executions determine the same input-output map. And, while each execution provides a different description of the network, they all correspond to the same MCN.

Descriptions of computational schemes will be considered equivalent if they determine the same input-output map. They will be considered structurally equivalent if, in addition, they determine the same MCN. Structural equivalence, which amounts to different choices of executions, leaves both the architecture and the function of the MCN unchanged. Other types of equivalence transformations will affect both the architecture and the function of the MCN but will keep its input-output map unchanged.

2.5 COMPARISON OF MCNs WITH OTHER NETWORK MODELS

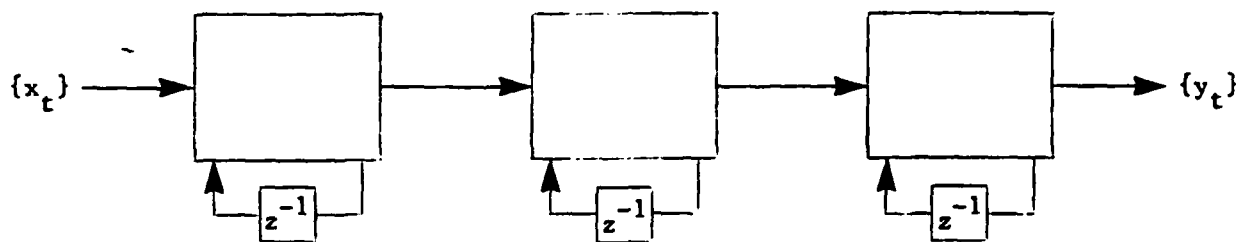
2.5.1 Block-Diagrams and Finite-State Machines

Numerical algorithms are most frequently described in terms of recursion equations involving indexed quantities, known as signals. Z-transform notation and block diagrams (or signal-flow-graphs) are sometimes used as equivalent descriptions of recursion equations.

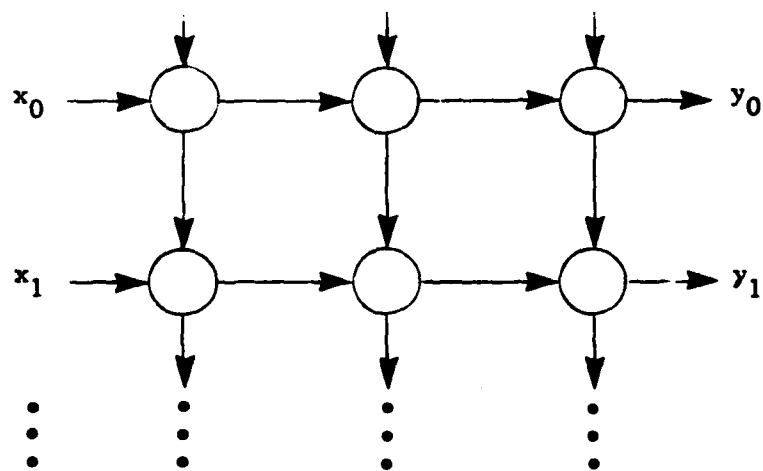
The main difference between MCNs and Z-transform block-diagrams is in the distinguished role of time in the latter model. A cascade connection of three blocks, each with its own state (Fig. 2-5a) corresponds to an MCN of infinite length (Fig. 2-5b). Each row of the MCN represents a single step of the recursion. Each input/output is a single variable, not a time-series. While the MCN description seems wasteful, it does in fact enhance our understanding of the various possibilities of implementation. Moreover, MCNs can describe irregular algorithms that cannot be described in terms of recurrence equations. This means that every block diagram can be converted into an MCN but not vice versa. The conversion amounts to duplicating the block diagram several times (once for every iteration of the recursion) and converting delay elements into direct connections between consecutive duplicates, as in Figure 2-5.

The preceding discussion considered only block-diagrams that correspond to sets of recursion equations. Such diagrams always consist of delay elements and memoryless operations. This means, of course, that only block-diagrams whose blocks represent finite-state machines can be converted in a straightforward manner into an MCN. Any other block-diagram has to be first converted into a state-space form (i.e., every block has to be represented by a state-space model or a combination of such models) before it can be converted into an MCN. Thus, in particular, any signal-flow-graph with rational transfer functions can be transformed into an MCN.

The correspondence between block-diagrams and MCNs provide a simple test for the executability (= computability) of algorithms represented by block diagrams.



a. Block-Diagram



b. Modular Computing Network

Figure 2-5. The Correspondence Between Block-Diagrams and MCNs

Executability Test

A finite block-diagram (or signal-flow-graph) whose blocks are characterized by delay elements and memoryless maps is executable if, and only if, the directed graph obtained by deleting delay elements from the diagram (or equivalently, by setting $z^{-1} = 0$ in the transfer functions) is acyclic.

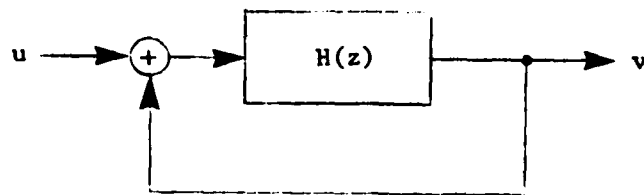
Proof:

Since delay elements are causal, they can never give rise to cycles in the corresponding MCN. In other words, since all operations in the i -th iteration temporally precede all operations in the $(i+1)$ -th iteration, the only cycles the MCN representation of a block-diagram may have must be contained within a single layer, corresponding to a single iteration. A single layer of the MCN is obtained by removing all delay elements from the block-diagram.

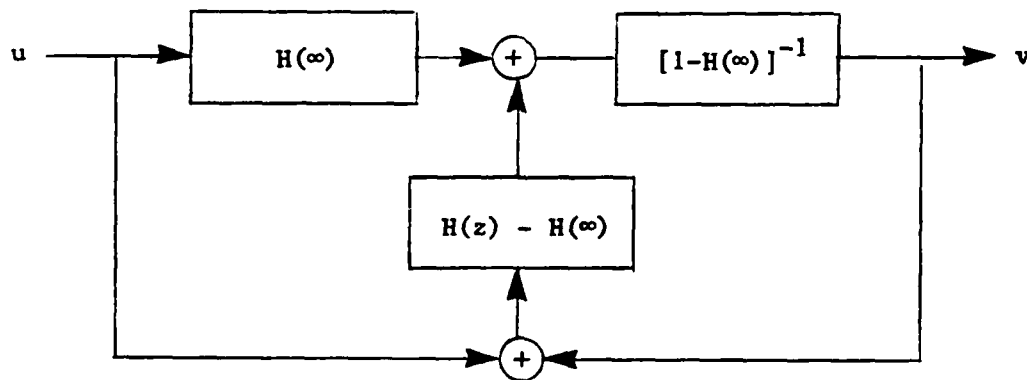
The test not only establishes the executability of a given block-diagram but indicates how to transform non-executable networks into executable ones. Consider, for instance, the network in Figure 2-6a. It is non-executable if $H(\infty) \neq 0$, because a cycle exists in the network for $z = \infty$. However, the same transfer function can be realized by the network in Figure 2-6b, which is executable.

2.5.2 Data-Flow-Graphs and Petri-Nets

The MCN is, clearly, a data-flow-graph [18] with the additional constraint that only one token is placed at every input of the network, and consequently, only one token eventually appears at every output of every processor. Thus, an MCN is safe by definition. In spite of this



a. Non-Executable Network



b. Equivalent Executable Network

Figure 2-6. Transformation of a Non-Executable Network into an Equivalent Executable One

observation every data-flow-graph (safe or unsafe) can be converted into an MCN, as long as every firing of a vertex in the flow-graph removes one token from every input line and adds one token to every output line. This constraint implies that the data-flow-graph can be converted into a block diagram involving only delay elements, advance elements and memoryless maps. This block-diagram can in turn be converted into a (not necessarily executable) MCN. The executability condition, when transformed back to the data-flow-graph domain becomes a cycle sum test, as described in [19].

Petri-nets are more general than data-flow-graphs. They allow two different kinds of vertices, known as places and conditions. Conditions correspond to our concept of processors, while places are combinations of multiple sources and sinks and thus have no counterpart in the MCN model. Petri-nets whose places have at most one input and at most one output are, in fact, data-flow-graphs (also known as marked graphs [20]) and can be converted into MCNs.

2.5.3 High-Level Programming Languages

Most high-level-language computer programs can be converted with little difficulty into MCNs. Each assignment statement of the program becomes a processor in the corresponding MCN. Program variables are mapped into network variables according to the following rules:

- (i) Each program variable, say x , is mapped into several network variables, denoted by x_1, x_2 , etc.
- (ii) An occurrence of a program variable x in the right-hand-side of an assignment statement is mapped into the same network variable x_i as the preceding occurrence of the same variable in the program.
- (iii) An occurrence of a program variable x in the left-hand-side of an assignment statement is mapped into a new network variable, i.e., into x_{i+1} if the most recent occurrence was mapped into x_i .

Recursions (do-loops) are mapped into sequential compositions of identical processors, each processor corresponding to one step of the recursions. The mapping of conditional recursions ('if' and 'while' statements) is somewhat more complicated and will not be described here. A separate technical memo will be devoted to the details of converting computer programs and other descriptions into MCNs, and vice versa.

The conversion of an MCN into a computer program is straightforward: Each processor is mapped into several assignment statements, and each network variable is mapped into a program variable.

2.5.4 Summary

The preceding analysis has shown that MCNs are essentially equivalent to computer programs, to block diagrams involving finite-state-blocks, and to a subclass of Petri-nets (marked graphs). The major distinction between MCNs and most other representations is the embedding of the notion of executability into the MCN model itself. Thus, the only way to design non-executable MCNs is by the introduction of cycles in the network architecture. Moreover, the test for executability is very easy to carry out and can be included in any compiler for MCN representations. It is much easier, on the other hand, to design malfunctioning Petri-nets or computer programs, and much more difficult to detect the errors in the design.

2.6 FORMAL LANGUAGE REPRESENTATION OF MCNs

To facilitate the application of the MCN model to both VLSI hardware design and software engineering it is necessary to develop a formal language version of the model, which preserves the convenience and simplicity of the graph-theoretic formulation. Such a formal language representation should not include more information than provided by the network graph. In particular, it should involve no details pertaining to the implementation of the MCN in a particular type of hardware. The matching between the requirements of an MCN model and the resources provided by a particular machine (e.g., sequential computer, dataflow computer, programmable

wavefront array) should be carried out by the compiler, not by the user/programmer. This will significantly simplify the coding phase of MCN models and eliminate most of the common programming errors.

To achieve the objective stated above the language should be capable of describing the two ingredients of the MCN model, variables and processors, and nothing else. It has to be a single assignment language, with each variable carrying its own name. Only two types of statements will be allowed: one for describing the interconnection between processors, the other for describing the functional characteristics of each processor. Regular interconnection patterns will be described by indexed loops. The sequential order of instructions in a program can be arbitrary and has nothing to do with the order of execution, which will be determined by the compiler in accordance with the precedence relation of the MCN, as well as the available storage and computing resources.

The purpose of imposing such severe limitations upon the syntax of the proposed language is to eliminate all flexibility in the translation of an MCN model into a computer program. Decisions about the structure of the MCN model for a given signal processing problem have to be made prior to the coding stage. Decisions about allocation of storage to variables and computing resources to computations have to be made after the coding stage, and preferably, by the compiler. This means that the coding stage itself can also be automated in the future, enabling the user to specify his designs interactively by 'drawing' the MCN model on a computer terminal.

Several languages have already been designed for modeling of parallel algorithms/architectures. Some of these focus upon the physical aspects of hardware implementation and almost completely lack the functional characteristics necessary to specify the algorithm. Others focus upon functional characteristics with little attention paid to structure. Only a few languages, like CRYSTAL [6], MDPL [7] and SIGNAL [23] maintain the balance between structure and function. Our approach combines ideas from these and several other languages with some unique concepts that emerged from the research on MCN models.

The principles underlying the construction of a formal language for MCN models are demonstrated by the following example

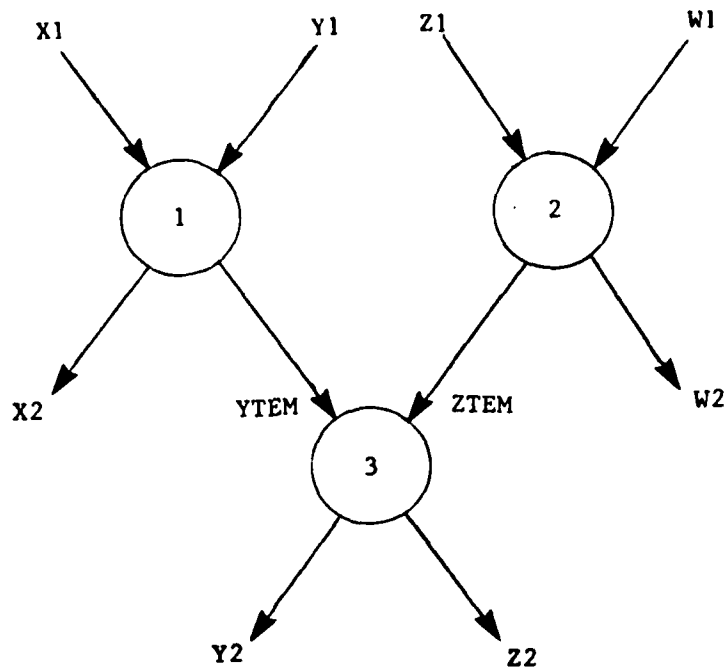


Figure 2-7. The MCN 'Example'

The corresponding MCN program is

```
MCN EXAMPLE (X1,Y1,Z1,W1; X2,Y2,Z2,W2)
```

```
BEGIN
```

```
  AM (X1,Y1;X2,YTEM)
```

```
  AM (Z1,W1;W2,ZTEM)
```

```
  AS (YTEM,ZTEM;Y2,Z2)
```

```
END EXAMPLE
```

```
PROC AM (X,Y;A,M)
```

```
BEGIN
```

```
  A:=X+Y
```

```
  M:=X*Y
```

```
END AM
```

```
PROC AS (X,Y;A,S)
```

```
BEGIN
```

```
  A:=X+Y
```

```
  S:=X-Y
```

END AS

The unique features of the language are:

- 1) Single assignment - each variable has its own name.
- 2) Modularity - each procedure is self contained and can be compiled and verified independently of the other procedures.
- 3) Hierarchy - there are three levels of specification: (i) Networks (MCN), which consist of another network or of atomic processors (PROC); (ii) Processors (PROC), which consist of assignment statements; and (iii) Variables, which may be of the types commonly used in conventional computer languages, and are used to construct assignment statements.
- 4) Information hiding - the components of the MCN program unit are specified only in terms of their inputs and outputs, without any details about their internal structure.
- 5) Modifiability and localization - the innards of every program unit can be modified without affecting the correctness of other units. The correctness of the modified unit can be tested without reference to other units.

Notice that the order of program units as well as the order of assignment statements in a processor is immaterial. This is made possible by the single assignment convention which associates one variable with every arc of the corresponding MCN graph. The names of processors and networks, on the other hand, can be duplicated to indicate identical inner structure (e.g., there are two 'AM' processors in the network 'EXAMPLE').

A formal language representation of an MCN model provides no information about the order in which the computations implied by the model will be executed. Following the precedence relation specified by the model, the computations can be arranged in layers, or wavefronts. The computations belonging to one layer can be executed in an arbitrary order, or even all in parallel. On the other hand, the execution of the $(i-1)$ -th layer must precede that of the i -th layer. The reformulation of the MCN model in terms of layers, which was introduced in Section 2.4, emphasizes the sequential-parallel nature of every MCN model, and serves as an intermediate step between the network-type character of the MCN model and the purely sequential nature of conventional computer languages. This wavefront representation also assists in determining variables that can use the same

storage area, and in allocating physical resources to computations when the number of available processors is less than that required for the fully parallel execution of a given layer. Thus, the transformation of an MCN model into a layered format plays a central role in the compilation of MCN programs for execution on a specified machine.

2.7 SUMMARY

A unified model for multilevel description and analysis of parallel algorithms and architectures has been developed. The model is general enough to describe any computational algorithm and to explicitly exhibit its parallelism.

The basic descriptive tool is a precedence graph, which indicates all possible implementations of the algorithm in either software or hardware. A simple structural condition (no cycles in the graph model) guarantees that the corresponding algorithm is executable. Different implementations of the same algorithm correspond to different orderings of the vertices (processing elements) of the precedence graph. Translation of software programs, data-flow graphs and z-transform descriptions of algorithms/architectures into precedence graphs and vice versa is easy to carry out.

The precedence graph approach clearly demonstrates the fact that storage (memory) requirements are determined by the implementation chosen for an algorithm rather than by the algorithm itself. Thus, the model of a single computing cell need not include storage at all. The most general cell is therefore a multiple input, multiple output map, viz.,

$$Y = f(U, \theta)$$

where U denotes inputs from other cells, θ denotes parameters, which may be locally stored in the cell, and Y denotes outputs. A cell is called:

linear - when f is linear in U (but not necessarily in θ)

time invariant - when parameters are time-invariant

Notice that a cell may be, in general, nonlinear and time-varying. However, it is always causal.

An actual hardware implementation of a cell involves also a delay of the output signal Y , consisting of a computation delay (the time required to compute Y once U is available) and a propagation delay (the time required for the output signal Y to reach its destination). The analysis of such delays and their effects upon the throughput of the MCN is presented in the following section.

SECTION 3

STRUCTURAL ANALYSIS OF MCNs

The notion of execution, defined in the previous section, provides several quantitative characterizations of the MCN architecture. In particular, it can be used to number the processors of an MCN and to introduce concepts of dimensionality. A refinement of the notion of execution leads to time schedules and to the formulation of composition rules for execution times. Thus, the objective of associating a unique execution time with every output of an MCN is achieved. The third objective, that of associating a unique measure of complexity with each MCN, has yet to be accomplished. Currently there is no consensus even upon the measure of complexity for a single processor, let alone for a network of processors. Some progress has been made in characterizing complexity in terms of 'area,' but more research is required before commonly-accepted rules for composition of complexity can be formulated. For this reason the topic of complexity will not be considered in the sequel.

3.1 NUMBERING OF VARIABLES AND PROCESSORS

The concept of execution, which was defined in Section 2.3, defines a numbering $E(x)$ on the variables of an MCN, viz.,

$$x \in S_i \iff E(x) = i \quad (3.1)$$

Since the partitioning $\{S_i\}$ and the numbering $E(\)$ determine each other and convey equivalent information, we shall call the function $E(\)$ itself an execution. Several variables may share the same value of $E(\)$, which means they belong to the same level S_i . If each level of an execution contains exactly one variable the execution is called sequential. The function $E(x)$ defines, in this case, a sequential ordering of the

variables and of the processors comprising the MCN. The numbering of variables determined by an execution $E()$ is consistent with the precedence relation since we clearly have

$$E(x) \geq 1 + \max_y \{E(y); y \in a(x)\} \quad (3.2)$$

Similarly, we can define a numbering of the processors by

$$E(p) := \max_x \{E(x); x \in I_1(p)\} \quad (3.3)$$

The value of $E(p)$ indicates the earliest instant at which all inputs of the processor p become available. We can also define a precedence relation for processors, viz.,

$$q \rightarrow p$$

if there exists a directed path from q to p . This relation, in turn, determines the ancestry set $a(p)$ of each processor by

$$a(p) := \{q; q \in P, q \rightarrow p\} \quad (3.4)$$

It can now be seen that an analog of (3.2) holds for the numbering of processors, viz.,

$$E(p) \geq 1 + \max_q \{E(q); q \in a(p)\} \quad (3.5)$$

Since a typical MCN has fewer processors than variables, the numbering of processors is a more convenient tool for structural analysis of an MCN.

3.2 DIMENSIONALITY AND ORDER

A family of sequential executions $\{E_i()\}$ on a given MCN is called representative if

$$q \in \alpha(p) \Leftrightarrow E_i(q) < E_i(p), \text{ all } i \quad (3.6)$$

Notice that a representative family can never consist of a single execution (except in the case of a purely sequential MCN) because there exist always two processors q, p such that $E(q) < E(p)$ even though q does not precede p (nor does p precede q). The following result shows that every MCN has at least one representative family.

Theorem 3.1

The collection of all sequential executions of a given MCN is a representative family.

Proof:

By the definition of execution

$$q \in \alpha(p) \Rightarrow E(q) < E(p)$$

for every execution (sequential or not). To prove the converse assume that $\{E_i(\)\}$ is the collection of all sequential executions, and that for some processors p, q

$$E_i(q) < E_i(p), \text{ all } i$$

Clearly p cannot precede q , but they may be incomparable. In this case there exists a non-sequential execution $\tilde{E}(\)$ such that

$$\tilde{E}(p) = \tilde{E}(q)$$

Since every execution can be transformed into a sequential one by arbitrarily ordering the variables in each level, it follows that \tilde{E} can be converted into a sequential execution, say E_0 , such that

$$E_0(q) > E_0(p)$$

This, however, contradicts the assumptions. Hence, p, q cannot be incomparable and we must have $q \leq \alpha(p)$.

A representative family with the smallest number of members will be called a basis (it need not be unique). The cardinality of bases is defined as the dimensionality of the MCN in consideration. The members of a basis $\{E_i(\cdot)\}$ define a coordinate basis for the network, such that the coordinates of a processor p are $\{E_1(p), E_2(p), \dots, E_n(p)\}$. Notice that the dimensionality of a network is bounded below by the dimensionality of all its subnetworks, so adding long chains of processors to a 2-dimensional network cannot reduce the overall dimension below 2 (Figure 3-1).

Every basis of an MCN determines a unique non-sequential execution obtained by ordering the processors according to the sum of their basis coordinates. For the example of Figure 3-1 this execution is

$$(1), (2,3) (4) (5) \dots (n)$$

The order of a basis is defined as the number of variables in the largest layer of the parallel execution determined by the basis. For the example above the order is 2 since there is a set of 2 processors in the parallel execution. Since an MCN may have many bases it has no unique order. Moreover, each execution E (not necessarily associated with a basis) has its own order, defined by

$$\text{ord}(E) := \max_i \{p; E(p) = i\} \quad (3.7)$$

Executions can be implemented in hardware by mapping each layer into a single iteration, with all the processors in the layer implemented in parallel. The order of an execution, which is the number of processors in the largest layer, is therefore a measure of the hardware complexity of such an implementation.

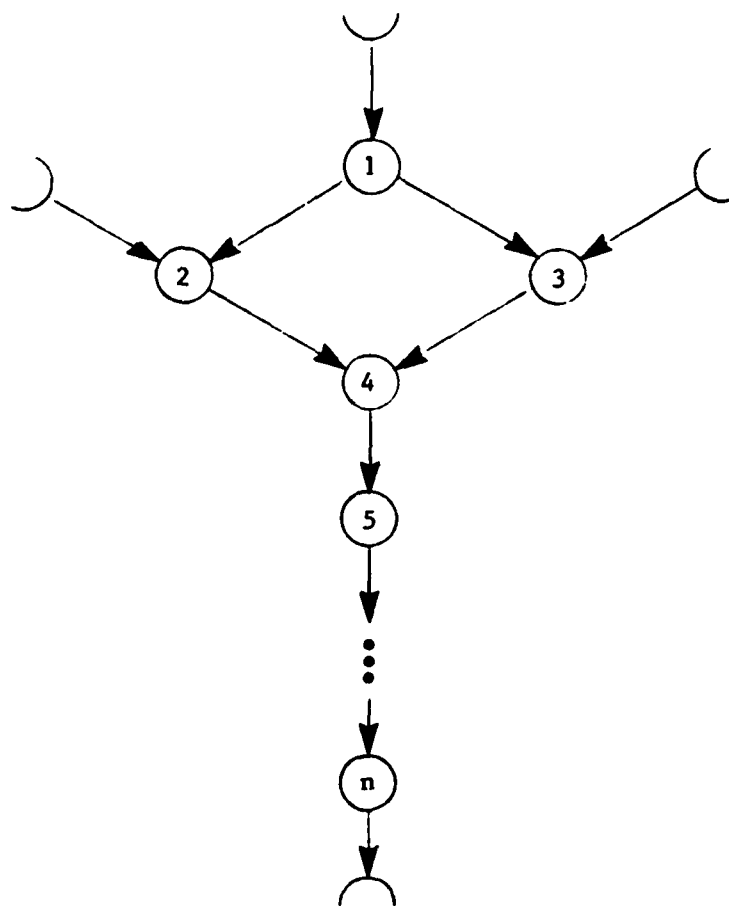


Figure 3-1. Example of a 2-D Network. The basis is formed by the executions 1,2,3,4,5,...,n and 1,3,2,4,5,...,n.

Once we have coordinate bases at our disposal we can apply metric arguments to the representation of an MCN. For instance, we can define distances between processors and introduce the concept of local communication between processors in a rigorous manner. However, more research is required to establish the properties of metrics defined by coordinate bases; in particular, it is not yet clear how the choice of the coordinate basis affects the metric.

3.3 SCHEDULES, DELAY AND THROUGHPUT

The execution of an MCN represents only its precedence relation and does not take into account the actual time required for execution. The evaluation of each variable requires a certain amount of execution time when implemented in hardware. Since each output of a processor may involve a different execution delay, execution times have to be specified for arcs of the precedence graph rather than for the vertices. The execution time associated with a variable x will be denoted in the sequel by $T(x)$. This is the time required to evaluate x from its immediate ancestors (= parents), i.e., from the variables that serve as inputs to the processor whose output is the variable x .

The incorporation of time delays into the notion of execution results in a schedule, which is formally defined as a function $\tau(x)$ that satisfies the constraint

$$\tau(x) \geq T(x) + \max_y \{\tau(y) ; y \in \alpha(x)\} \quad (3.8a)$$

and is zero for the network inputs, viz.,

$$x \in I_1(P) \Rightarrow \tau(x) = 0 \quad (3.8b)$$

This constraint guarantees, in particular, that the parents of x will be available at time $\tau(x)$. Thus, schedules are refinements of executions. In particular, with every execution $E(\)$ we can associate a schedule $\tau(\)$ by choosing

$$\tau(x) = \max_y \{\tau(y) + T(x) ; E(y) = E(x) - 1\} \quad (3.9)$$

Such schedules are, generally, non-minimal in the sense that some operations have all their inputs available before their scheduled execution time, i.e., (3.8) holds with a strict inequality for such operations. A schedule which satisfies (3.8) with equality for every $x \in X$ is called minimal.

Minimal schedules are important because they characterize the fastest executions of a given MCN. This property is made explicit by the following result.

Theorem 3.2

Every structurally finite MCN has a unique minimal schedule $\hat{\tau}(\)$. The minimal schedule satisfies

$$\hat{\tau}(x) \leq \tau(x) \quad (3.10)$$

for every $x \in X$ and for every schedule $\tau(\)$.

Proof:

Since by Theorem 2.1 a structurally finite MCN has a countable number of variables, the result can be established by induction. Thus, let S be a subset of X that is closed under the ancestry relation, namely for every $x \in S$ we must have $\alpha(x) \subset S$. Assume that S has already been assigned a minimal schedule $\hat{\tau}(\)$ and that this schedule also satisfies (3.10).

Choose a variable y not in S and consider the augmented network determined by $S \cup \alpha(y)$. We need to show that $\hat{\tau}(\)$ can be extended to this augmented network and that it will satisfy both (3.8) and (3.10). The schedule $\hat{\tau}(\)$ is now extended to $\alpha(y)$ in the following manner:

- (i) Assign $\hat{\tau}(z) = 0$ to every $z \in \alpha(y)$ that has no ancestors.
- (ii) Identify the collection of variables for which all ancestors have already been assigned a schedule (this set is never empty). Assign to each one of these variables the schedule

$$\hat{\tau}(z) := T(z) + \max_w \{\hat{\tau}(w); w \in \alpha(z)\}$$

For every $w \in \alpha(z)$, either $\hat{\tau}(w) = 0$ or $w \in S$, so that $\hat{\tau}(w) \leq \tau(w)$ for any schedule $\tau(\cdot)$. Since any schedule $\tau(\cdot)$ has to satisfy (3.8) we obtain

$$\begin{aligned}\tau(z) &\geq T(z) + \max_w \{\tau(w); w \in \alpha(z)\} \\ &\geq T(z) + \max_w \{\hat{\tau}(w); w \in \alpha(z)\} = \hat{\tau}(z)\end{aligned}$$

which proves that (3.10) is preserved in this step.

(iii) Augment S , viz.,

$$S := S \cup \alpha(y)$$

and go back to (ii).

The repeated application of this procedure results in the assignment of $\hat{\tau}(x)$ to every variable of the MCN. The resulting schedule is minimal, i.e., it satisfies (3.8) with an equality, unique (by construction) and also satisfies (3.10).

As with executions, we can also define schedules for processors. The schedule of a processor $p \in P$ is defined as

$$\tau(p) := \max_x \{ \tau(x); x \in X_1(p) \} \quad (3.11)$$

in analogy with 3.3. It is the instant at which all input variables of the processor become available. Some of the inputs of the processor may become available earlier and need, therefore, storage or buffering until they can actually be used. A variable x is called critical with respect to a given schedule $\tau(\cdot)$ if

$$x \in X_1(p) \Rightarrow \tau(x) = \tau(p) \quad (3.12)$$

and non-critical otherwise. Thus, the schedule of each processor is determined by the schedule of its critical inputs. Since non-critical variables require storage the general objective of scheduling is to reduce the total storage requirements.

Storage is measured by the product of volume (e.g., the number of bits to be stored) and duration. The duration of storage for a variable $x \in X_1(p)$ is the difference between the time it becomes available and the most recent instant it still needs to be available, i.e.,

$$\max \{ \tau(y) - T(y); y \in X_0(p), x \rightarrow y \} - \tau(x)$$

This interval will be minimized if we choose the difference $\tau(y) - T(y)$ as short as possible. In view of (3.8), we have to choose $\tau(y) - T(y) = \tau(p)$, namely the minimal schedule also minimizes the storage requirements of the network. The minimal schedule still has both critical and non-critical variables. However, only the critical ones determine the schedule, as demonstrated by the following result.

Lemma 3.3

Every processor in a structurally finite MCN is connected to a network input by a finite path whose variables (arcs) are critical under the minimal schedule.

Proof:

The definition of a critical variable implies that every processor has at least one critical input variable. The critical path is obtained by tracing back through the critical inputs of the preceding processors. Since the ancestry of each processor is finite, this procedure terminates in a finite number of steps when the path reaches a network input.

Corollary 3.3

The minimal schedule of a processor equals the length (sum of processing delays) of a critical path that connects a network input to this processor.

The corollary implies an interesting principle for the physical design of hardware implementations--critical paths need to be considered first so that the length of the physical connections along the path can be minimized. Non-critical paths can accommodate extra propagation delays and can, therefore, be designed later.

The construction of a schedule is based upon the assumption (3.5b) that all MCN inputs are available at the very beginning. Thus, a zero schedule was assumed in (3.8) for every MCN input, i.e.,

$$x \in X_1(P) \implies \tau(x) = 0$$

This is, however, inessential, since many of these inputs will not be required until much later. The scheduling of the network inputs can be modified, once a schedule $\tau(\cdot)$ has been determined, to reflect the earliest instant they are required in the execution. Thus, for every $x \in X_1(P)$ redefine the schedule of the inputs to be

$$x \in X_1(P) \implies \tau(x) := \tau(p) \quad \text{where } x \in X_1(p) \quad (3.13)$$

and no buffering, or storage, of the inputs will be necessary. This is particularly important if not all the inputs can be made available in the same instant, e.g., in real time processing of time-series. Notice that this modification in the scheduling of inputs does not affect the schedule of any other variable in the network. This is so because only non-critical input variables are adjusted. The meaning of (3.13) is that all network inputs are made critical to reduce the storage requirements of the network.

The schedule of output variables is commonly known as delay. The delay of x is the time that has elapsed from the moment some variable in $a(x)$

becomes available until the moment the variable x itself becomes available. This is, clearly,

$$\tau(x) = \min \{ \tau(y); y \in u(x) \}$$

and in many cases it will be equal to $\tau(x)$. In typical signal processing applications the delay of outputs usually increases without limit as more and more inputs are applied to the processor and more and more outputs are evaluated. In such cases one is interested in the rate of output evaluation, commonly known as throughput, rather than in the delay of the outputs. The throughput is roughly the number of MCN outputs that are evaluated in a unit of time. Since this rate may vary, we need a more rigorous definition based on the concept of schedule.

Every schedule determines a temporal ordering of the MCN variables (it need not be sequential), which is consistent with the precedence relation between variables. In order to quantify the rate at which output variables are evaluated, we define the output counting function

$$N_o(\tau) := \text{number of elements in the set} \quad (3.14)$$

$$\{y; y \in X_o(P), \tau(y) < \tau\}$$

The input counting function can be similarly defined, viz.,

$$N_i(\tau) := \text{number of elements in the set} \quad (3.15)$$

$$\{y, y \in X_i(P), \tau(y) < \tau\}$$

We can now plot the counting function $N(\tau)$ as a function of τ for both the inputs and the outputs (Figure 3-2). The functions $N_i(\tau)$; $N_o(\tau)$ are, of course, staircase functions (indicated by broken lines in Figure 3-2) and can be upper-bounded by a pair of continuous, piecewise-linear curves (indicated by the solid lines in Figure 3-2). The slope of these

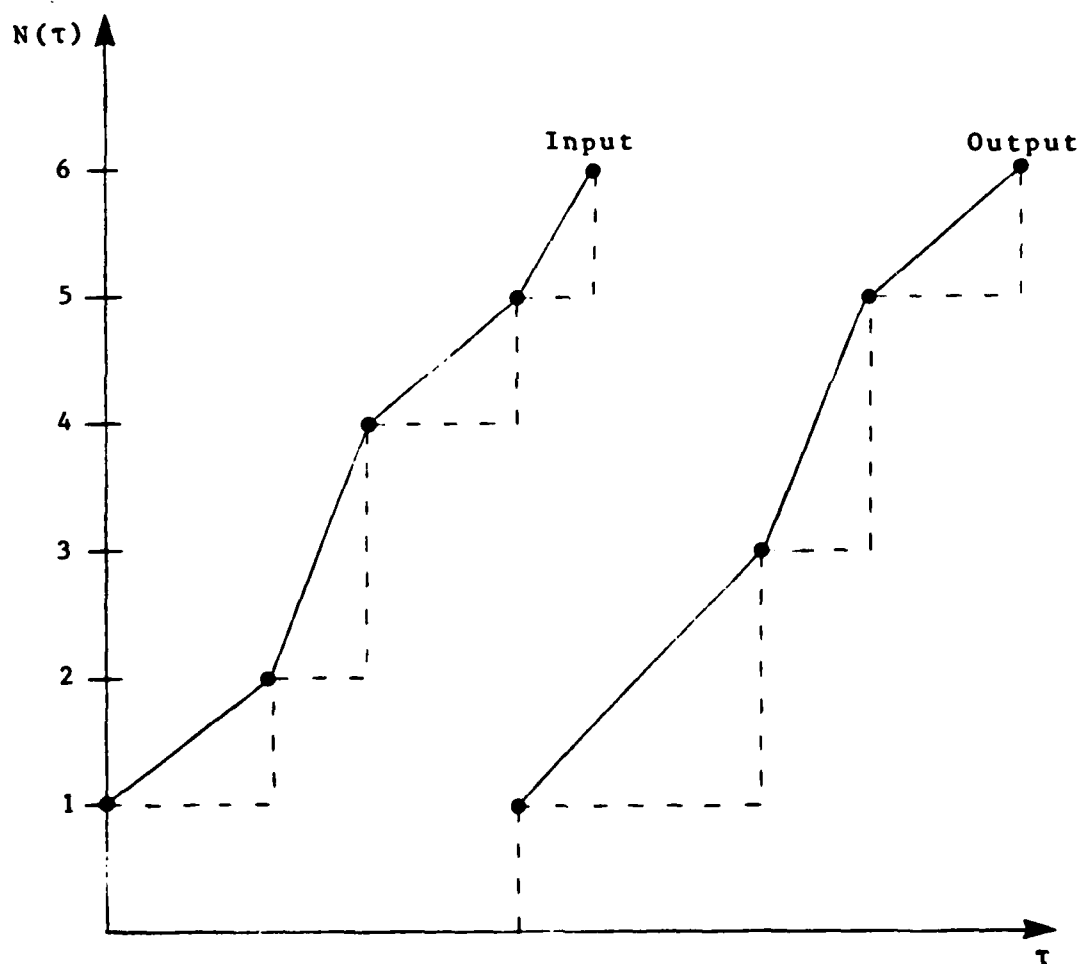


Figure 3-2. Input and Output Throughputs of an MCN.

curves (which are always strictly monotone increasing) is a measure of the rate of information flow into the network and out of it, and will be called the input and output throughput, respectively. A schedule is called regular when both its input and output throughput are periodic with the same period (and, in particular when both throughputs are constant). An MCN is called temporally-regular when its minimal schedule is regular. Many temporally-regular networks have equal input and output throughputs, but this need not be true, in general.

3.4 SPACE-TIME DIAGRAMS

The continuous-time character of the schedule is best demonstrated by introducing a time-axis into the graphical description of an MCN. The vertices are arranged so that the vertical displacement from the top of the diagram to the location of any given vertex p indicates, on an appropriate scale, the value of the schedule $\tau(p)$ for this vertex (Figure 3-3, compare with Figure 2-1). This space-time diagram has several interesting properties:

- (1) All arcs point downward.
- (2) The vertical displacement of an arc indicates the total execution time associated with this operation, including any buffering time that may be required beyond the actual execution time $T(x)$.
- (3) Changes in local execution times are easily accounted for by shifting the corresponding vertices up or down along the time scale. The global effects of such shifts are clearly depicted by the diagram.
- (4) Non-executable MCNs (with zero or negative execution times) can still be described by the diagram. This is useful to establish equivalence between various descriptions of the same MCN (e.g., precedence graphs and signal flow graphs).

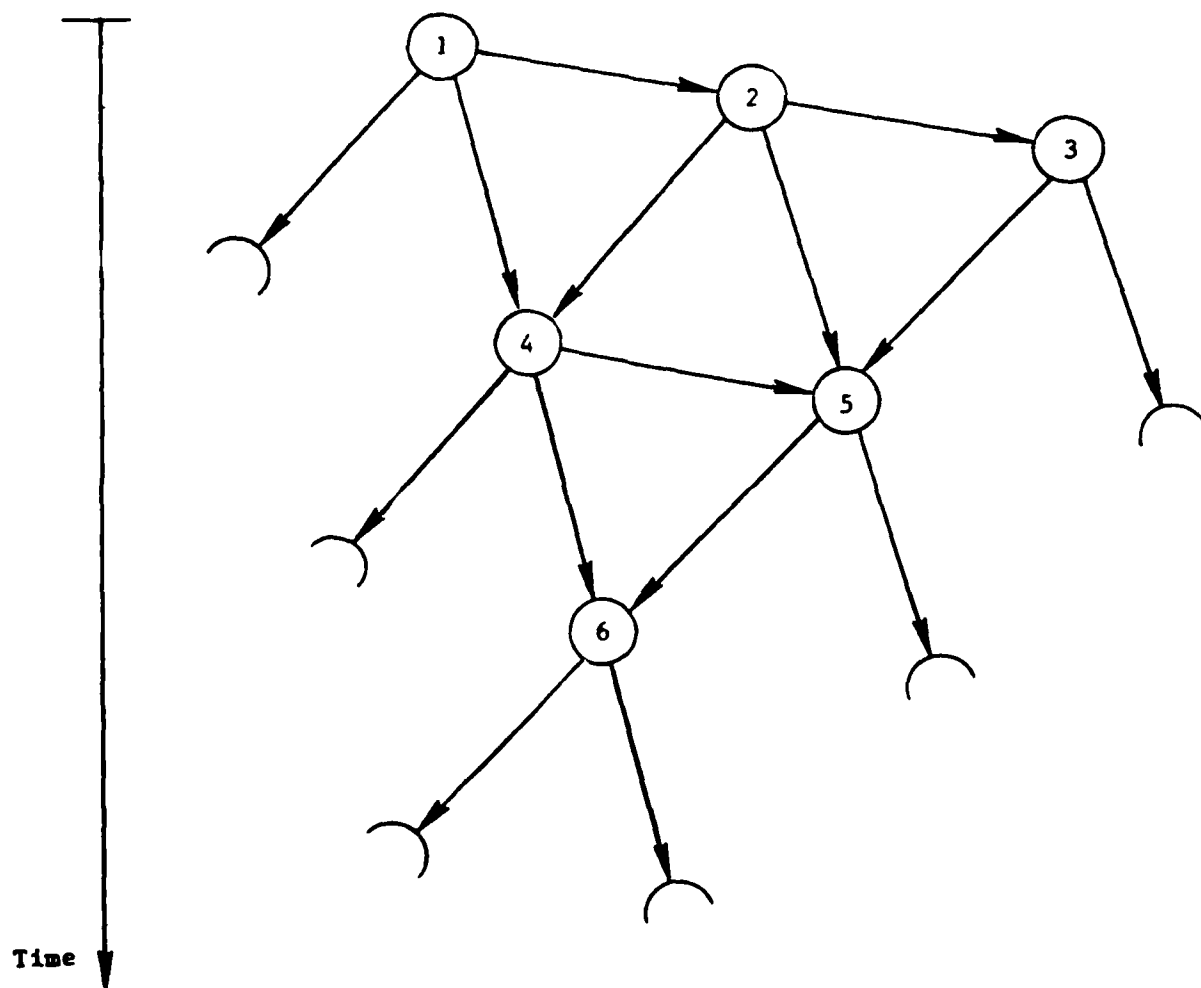


Figure 3-3. Introduction of a Time Scale into the Architecture

The collection of processors (vertices) with the same schedule form an isochrone.

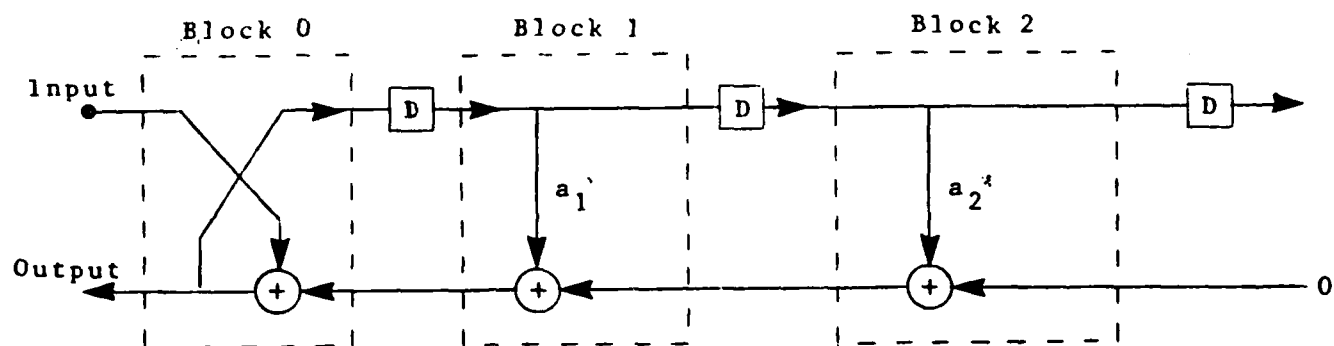
The execution of a network according to a given schedule may now be interpreted as the propagation of a single wavefront of activity through the architecture. The location of the activity wavefront at any given instant is indicated by the corresponding isochrone. Observe that the isochrones are parallel straight lines (or parallel planes if the precedence graph is described in a three dimensional space) and do not intersect. Also notice that the inputs and outputs of a temporally-regular network are evenly distributed in time (i.e., along the vertical axis of the space-time diagram). These properties are particularly significant for the analysis of iterative MCNs, which is carried out in Section 4.

As an illustration of the equivalence between various descriptions of the same MCN consider the block-diagram of an IIR filter (Figure 3-4a). The corresponding MCN (Figure 3-4b) can be rearranged in many ways without modifying the architecture of the network. However, if Figure 3-4b is interpreted as a space-time diagram (with time being the vertical axis), such modifications result in different schedules and also in different block-diagrams. In particular, the delay elements can be moved to the lower path (Figure 3-5) or split between the two signal paths (Figure 3-6). The latter version is the only one that can be implemented in hardware because it contains only downward-pointing arrows; the other two versions require instantaneous evaluation of each variable associated with a horizontal arrow. The third description makes it also clear that the time interval between successive application of inputs is equal to two delay units. It is also possible to associate unequal computing times with the forward and backward propagation through each block. After all, the forward path only feeds information through the block while the backward path involves a multiply-and-add operation. The resulting space-time diagram (Figure 3-7) has delays T_f , T_b associated with the forward and backward paths, and the input interval is clearly $T_f + T_b$. Notice that the block diagram description involves two different delay blocks: This is known as a multirate implementation [8]. The throughput rates are, nevertheless, equal to $(T_f + T_b)^{-1}$ for both the input and the output.

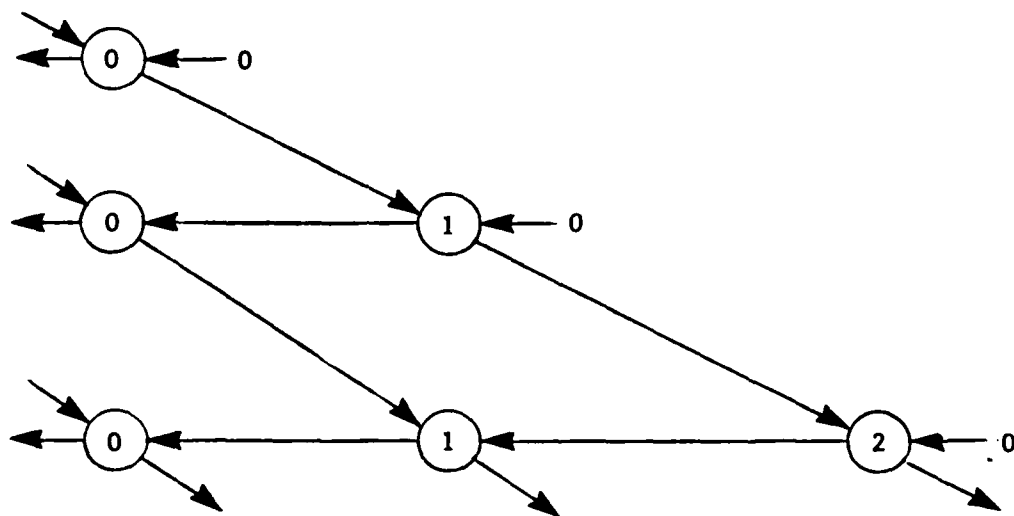
The same technique can be applied to analyze the several proposed systolic-array-like implementations for matrix multiplication: the

hexagonal array of H.T. Kung [5], the improved hexagonal array of Weiser and Davis [4], the wavefront array processor of S.Y. Kung [7] and the direct form realization of S. Rao [10]. Details are provided in Appendix E.

The analysis of the previous examples makes it clear that the common MCN architecture shared by all the representations of a given processing system induces certain invariants. For instance, the total number of outputs of each processor remains invariant, even though in some representations some of these outputs are connected to a local memory rather than to a nearby processor (Figure 3-8). The same is true for the total number of inputs of each processor. Notice that the blocks in Figure 3-8a are still the same as in Figure 3-4a, including the orientation of paths (one forward, one backward). On the other hand, the roles of the blocks are quite different; in particular, outputs are obtained from the local memories rather than from the left-most block alone, as in Figure 3-4a.

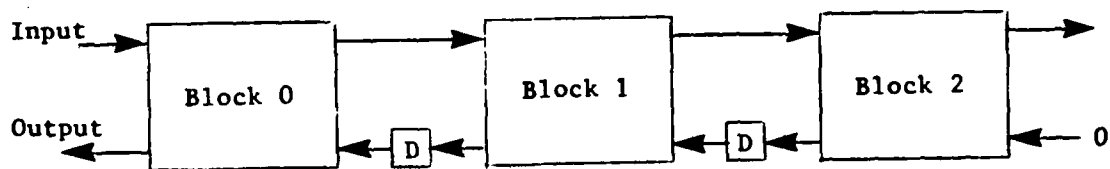


a. Block-diagram

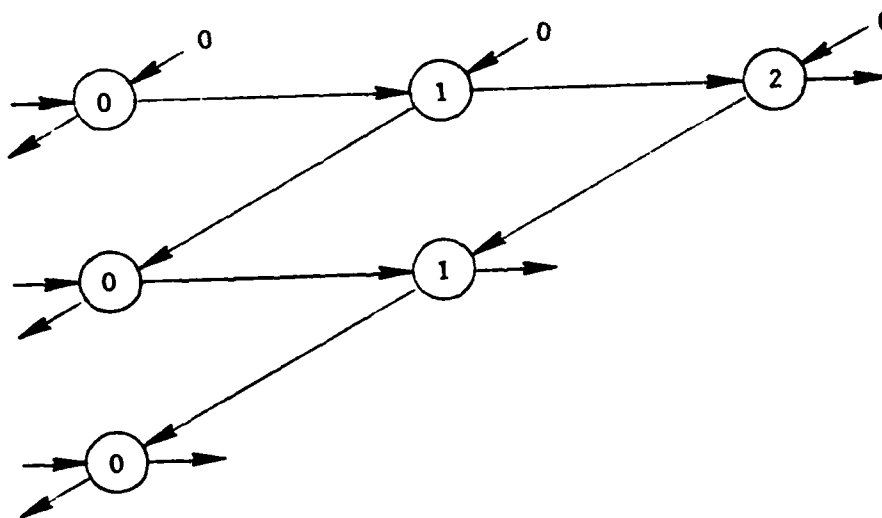


b. Space-time diagram

Figure 3-4. Schematic Description #1 of an IIR Filter

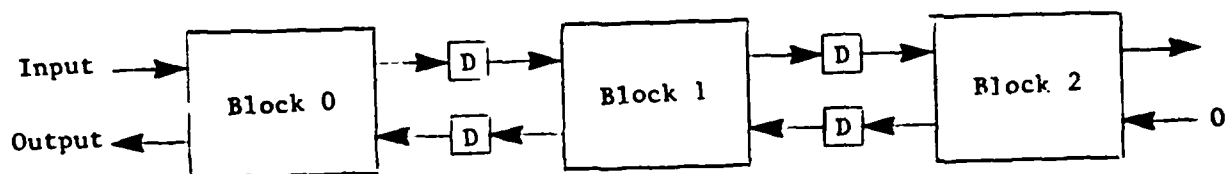


a. Block-diagram

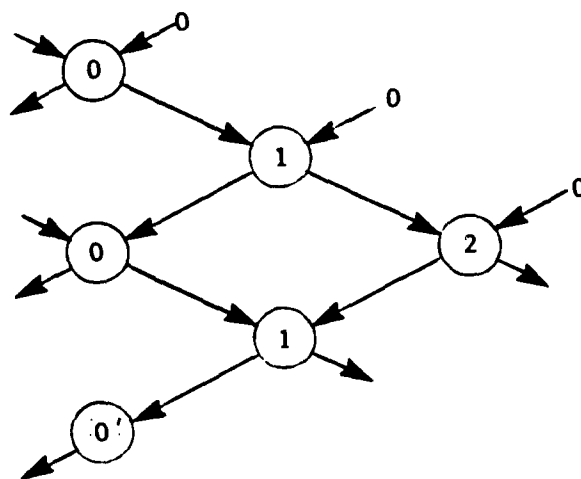


b. Space-time diagram

Figure 3-5. Schematic Description #2 of IIR Filter

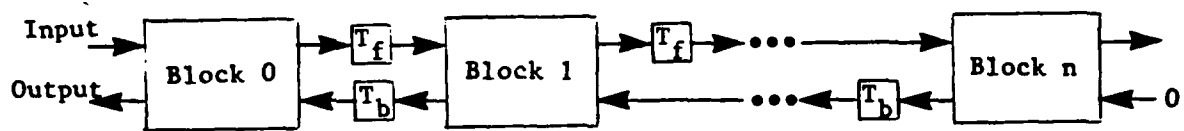


a. Block-diagram

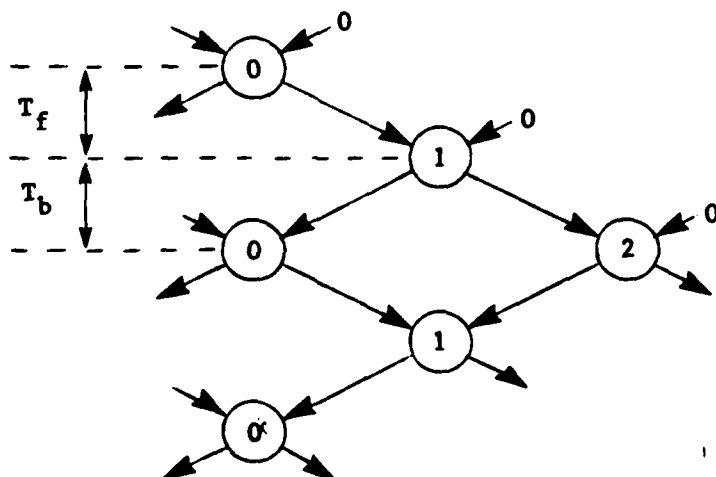


b. Space-time diagram

Figure 3-6. Schematic Description #3 of an IIR Filter

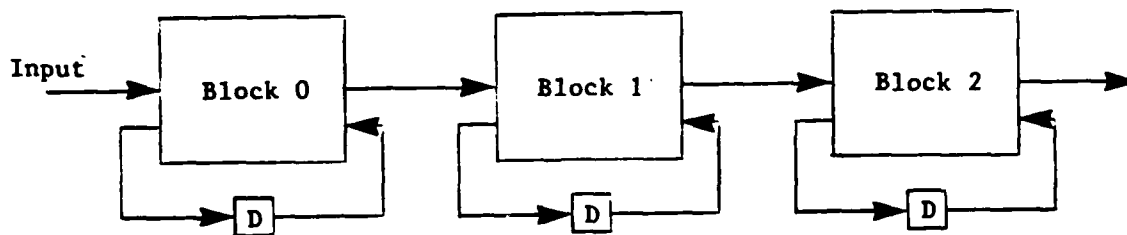


a. Block diagram

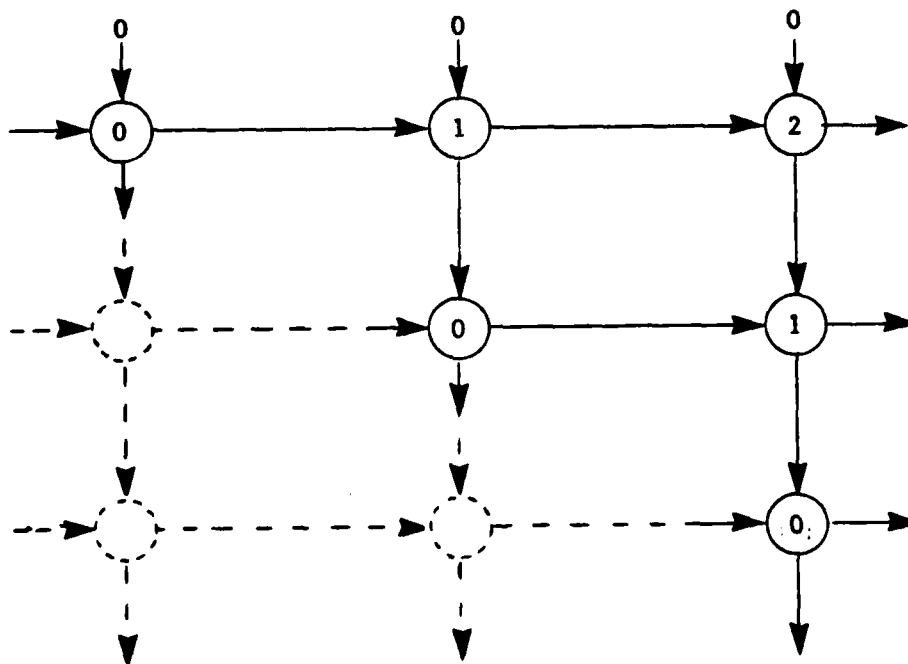


b. Space-time diagram

Figure 3-7. Multirate Implementation of an IIR Filter ($T_f < T_b$)



a. Block diagram



b. Space-time diagram

Figure 3-8. Schematic Representation of IIR Filter Involving Local Memory

3.5 SUMMARY

Techniques for analysis of pipelinability, schedules, and throughput in systolic-array-like configurations have been developed based on the MCN representation of parallel algorithms/architectures. Architecture evaluation was also based on graph theoretic properties of the MCN model: The dimensionality, degree of parallelism, and throughput of a given architecture are all determined by analysis of its precedence graph.

The major difficulty in the analysis of computing networks lies in the translation of low-level input-output relations to high-level ones, and vice versa. We have shown that the problem reduces to the factorization of the global (high-level) input-output map into a product of purely parallel maps, corresponding to the concept of wavefront propagation in the network. More specifically, the global input-output map is a sequential composition of the maps corresponding to the layers of an execution.

SECTION 4

ITERATIVE AND COMPLETELY REGULAR NETWORKS

4.1 ITERATIVE MCNs AND HARDWARE ARCHITECTURES

An MCN is called iterative when it can be described as a sequential composition of identical subnetworks, i.e.,

$$\mathcal{C}_{\text{network}} = \mathcal{C} * \mathcal{C} * \dots * \mathcal{C}$$

Each of the identical components \mathcal{C} will be called an iteration. One reason for this name is that the MCN can be executed by implementing a single component \mathcal{C} in hardware and simulating a sequential composition of such components by spreading the execution of the components in time. The motivation for studying iterative MCNs is that most signal-processing algorithms and, in particular, all systolic-array-like architectures can be described by such networks. Observe that every block-diagram representation corresponds to an iterative MCN. The iterative structure induces certain regularity constraints upon the MCN which lead to a simplified representation.

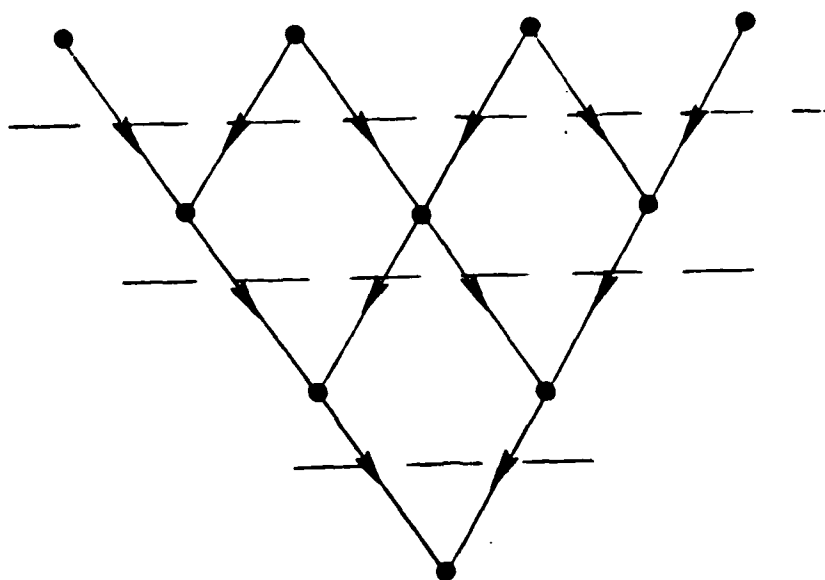
The minimal schedules of iterative networks are, clearly, periodic with the same period for input and output schedules. Thus iterative MCNs are temporally-regular. In addition, they are functionally-regular in the sense that each iteration involves the same function \mathcal{F} . Consequently, their properties can be determined by analyzing a single iteration. For instance, the entire network is acyclic (hence executable) if a single iteration is acyclic. In particular, the executability of z-transform representations of iterative MCNs is tested by removing all separators and examining the remaining directed graph for occurrence of cycles (see also [9]). Similarly, the (minimal) schedule of the network can be determined by considering a single iteration.

Iterative MCNs are commonly described by recursion equations (or equivalently by z-transform diagrams), data-flow diagrams (marked graphs), or by 'do-loops' in high-level programming languages. While precedence graphs of iterative networks still indicate all possible executions, recursion equations restrict the choice of execution to one or at most two possibilities (Figure 4-1). And while precedence graphs avoid the pitfall of non-executable iteration by explicitly describing each iteration as part of an executable (acyclic) precedence graph, data-flow diagrams contain cycles which may cause the entire MCN to be non-executable.

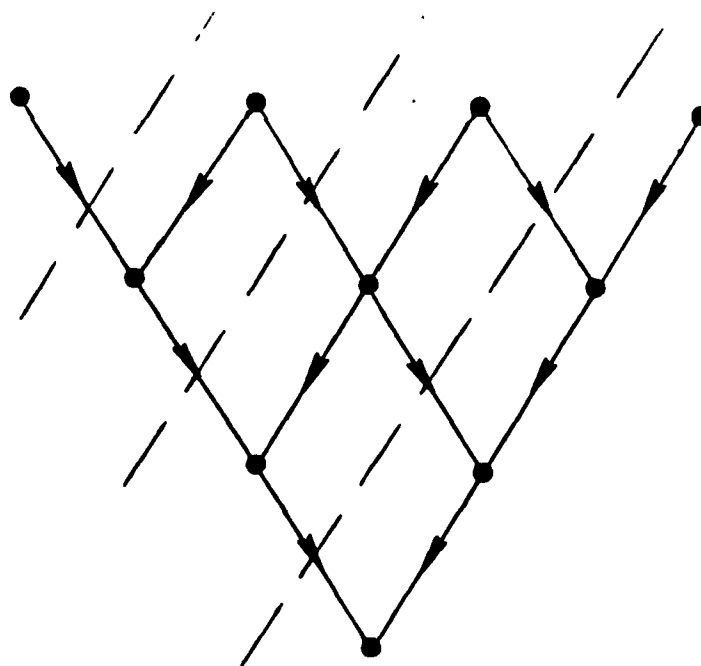
Since all iterations are identical, the schedules of every two consecutive iterations differ by the same constant, which we shall call the input interval. The input interval is the period of the input schedule or, equivalently, of the input throughput, as well as of the output schedule (recall that iterative MCNs are temporally regular). It determines an upper bound on the rate at which inputs are applied to the network (lower rates are permitted, but require additional buffering).

The time-space diagram of an iterative MCN corresponds to its minimal schedule and is, therefore, periodic. It is important to notice that the period (= input interval) is, in general, shorter than the time required to complete the execution of a single iteration (= the iteration delay). This means that hardware realizations of the MCN can be pipelined: New inputs can be applied before the processing of previous inputs has been completed.

The functional regularity of iterative MCNs implies that they can be implemented in special purpose VLSI hardware by mapping the precedence graph of a single iteration directly into silicon. Each processor is mapped into a cell ('processing element') and precedence relations are mapped into interconnections between cells. Neither translation nor hardware compilation are required to accomplish this mapping since the hardware architecture is an exact image of a single layer of the network architecture. An execution is now interpreted as the propagation of a sequence of wavefronts through the hardware rather than the propagation of a single activity wavefront through the iterative MCN (Figure 4-2). The time spacing of these wavefronts equals the period of the underlying MCN minimal schedule.



a. Parallel Input Application



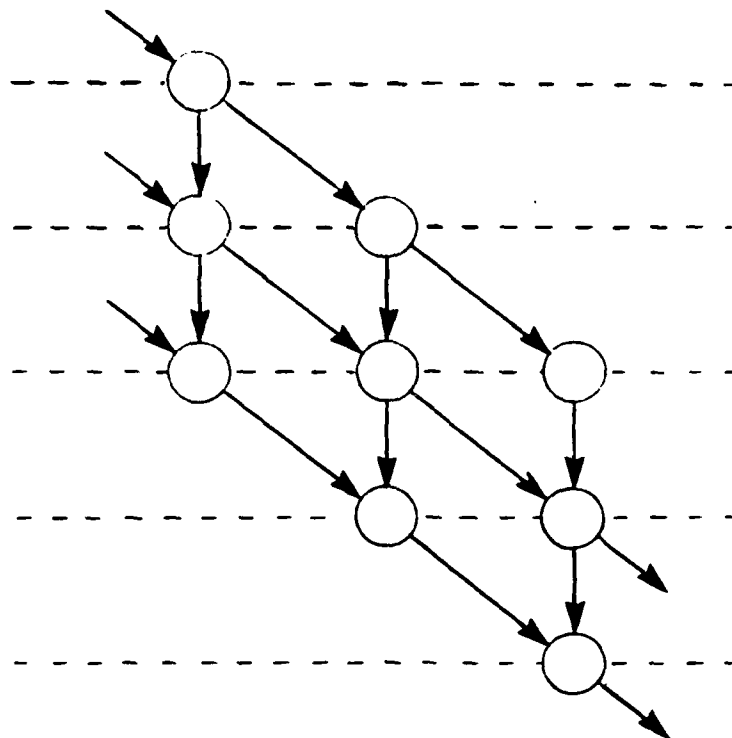
b. Sequential Input Application

Figure 4-1. Equivalent Pipelined Executions of an MCN.

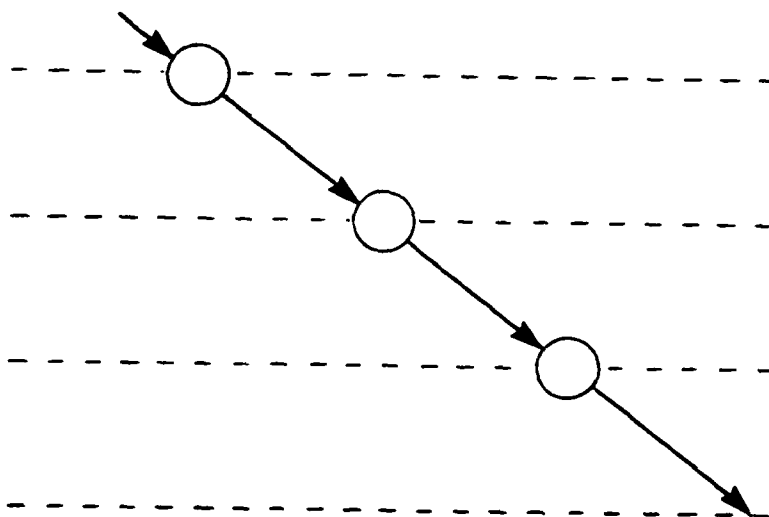
Since a single layer of the MCN is used to 'simulate' the entire network each processor is activated many times and each arc of the hardware architecture corresponds to a time-series of variables rather than to a single variable. This raises a design problem of a new kind: It is necessary to guarantee that variables do not disappear before they have been used to evaluate their successors. There are three different solutions to this problem:

- (1) Iterative execution: A new iteration is initiated only after the execution of the preceding iteration has been completed. This means that the input interval is extended (by buffering of intermediate results) to the length of the iteration delay, and the time-overlap between iterations is completely eliminated.
- (2) Scheduled execution: The (minimal) schedule of the network is determined in advance and execution is carried out according to schedule. Buffering is provided to guarantee the availability of inputs to processors on schedule (only non-critical variables need to be buffered).
- (3) Self-timed execution: Processors are activated as soon as their inputs become available. Acknowledgment signals ('hand-shaking') are used to guarantee the correct transfer of data between processors.

While scheduled execution offers the shortest execution time and requires a fairly simple control system, it is extremely sensitive to scheduling perturbations. Such perturbations, which are caused by clock-skewing and local variations in execution times, may result in loss of synchronization between cells and a complete failure of the system. Iterative execution is insensitive to scheduling perturbations and requires a very simple control system, but wastes processing time since the hardware is idle most of the time. Self-timed execution provides a nice tradeoff between these two extremes: Its execution time is only slightly longer than the theoretical minimum achieved by scheduled execution; and the control system it requires has about the same complexity as the timing system for scheduled execution. It is interesting to observe that the conditions for self-timed execution [16], [17] coincide with the concept of admissible composition, which was shown to be the necessary and sufficient condition for executability in general. Thus, every MCN can be implemented as a self-timed system.



a. MCN Perspective



b. Hardware Architecture Perspective

Figure 4-2. Execution Interpreted as Activity Wavefront Propagation

The notion of self-timed execution suggests the introduction of self-timed block-diagrams. These are obtained by removing the delay-elements from a conventional block-diagram and replacing them by direct connections. The hardware implementation of such self-timed diagrams is straightforward provided two simple rules are obeyed:

- (i) Each cell is activated as soon as all its inputs become available and deactivated as soon as all its outputs have been evaluated.
- (ii) Each input variable is accompanied by an acknowledgment line. Each input port (sink) acknowledges the arrival of a new input variable to the processor that evaluated this variable. The acknowledgment is sent when the processor connected to the input port becomes activated.

These rules assume that each cell is provided with sufficient memory to store its output variables until they become acknowledged.

The acknowledgment of inputs associated with self-timed implementations can (and should) be reflected in the space-time diagram of the network. Acknowledgment signals are just one more set of variables in the network, and are represented in the space-time diagrams by arcs, as any other variable. For instance, a cascade connection of (identical) processors (Figure 4-3a) has an input interval of $\tau_1 + \tau_2$ where τ_1 is the execution time of the processor and τ_2 is the delay between the reception of an acknowledgment signal from the subsequent processor and the transmission of an acknowledgment signal to the preceding processor (Figure 4-3b). The interval τ_2 includes the propagation time through the processor and the connecting wires plus the time required to carry out checks on the input data (parity, error detection, fault detection, etc.). Notice that the need for explicit acknowledgment can be eliminated in many cases, e.g., when there is an information carrying path along the cascade in the backward direction.

The horizontal dimension of space-time diagrams can now be interpreted as hardware: Processors located along the same horizontal line (isochrone) represent computations that need to be carried out



a. Self-Timed Block-Diagram

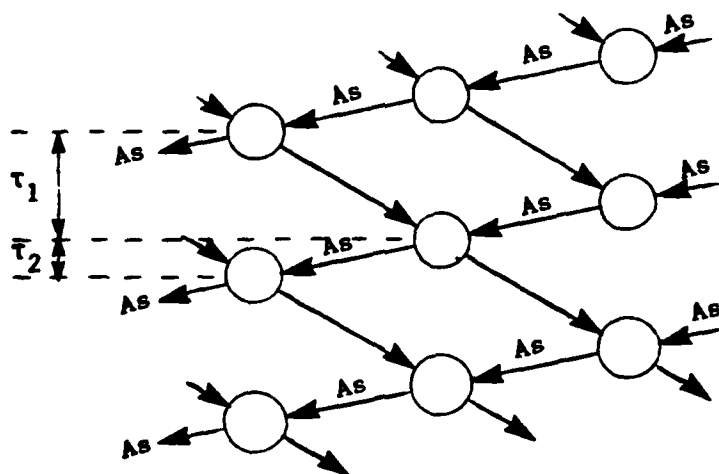
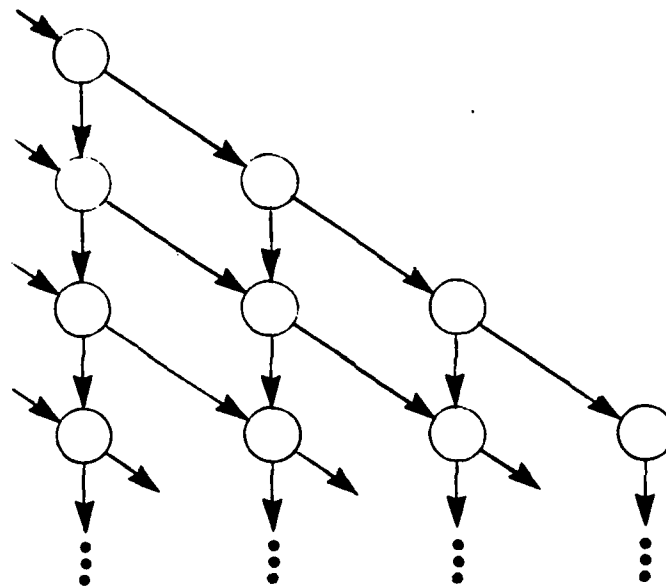


Figure 4-3. Propagation of Acknowledgment Signals (AS) in Self-Timed Systems

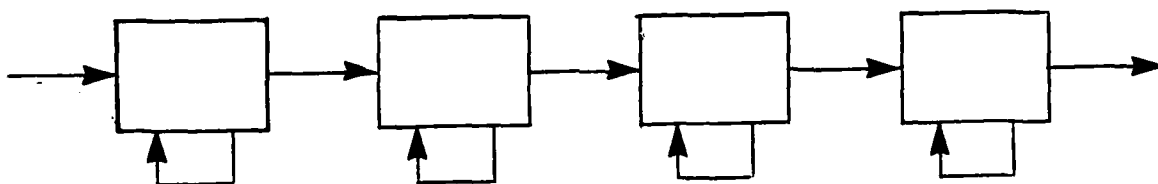
simultaneously and must, therefore, be implemented in parallel hardware. We shall adopt the convention of interpreting the vertical dimension of space-time diagrams as pure time: Processors located along the same vertical line will represent computations that are carried out by the same processing element but during different (non-overlapping) intervals of time. Thus, for instance, the MCN of Figure 4-4 can be implemented in hardware with four processing elements (Figure 4-4b). Each vertical column of processors in the space-time diagram of the MCN (Figure 4-4a) is mapped into a single hardware cell; connections between columns are mapped into physical connections between cells and connections within columns are implemented by locally storing intermediate results inside the appropriate cells.

Self-timed or scheduled execution is, indeed, faster than iterative execution only if the input interval is shorter than the execution time of a single iteration. An implementation of such an execution will initiate a new iteration before the execution of the previous iteration has been completed. Such implementations deserve to be called pipelined. Thus, iterative executions are never pipelined, while self-timed (or scheduled) executions are pipelined only for pipelinable MCNs.

Notice that the input interval is uniquely defined for every temporally-regular MCN, but the iteration delay (= execution time of a single iteration) depends upon the partitioning of the MCN into iterations. Since this partitioning need not be unique, an iterative MCN may have several hardware realizations, each with a different iteration delay. Thus, pipelining is primarily a property of a given hardware realization. An MCN is considered pipelinable if it has at least one pipelined realization. Pipelinability is most frequently associated with completely regular MCNs (= systolic-array-like networks). The connection between complete regularity and pipelinability is discussed in the following section.

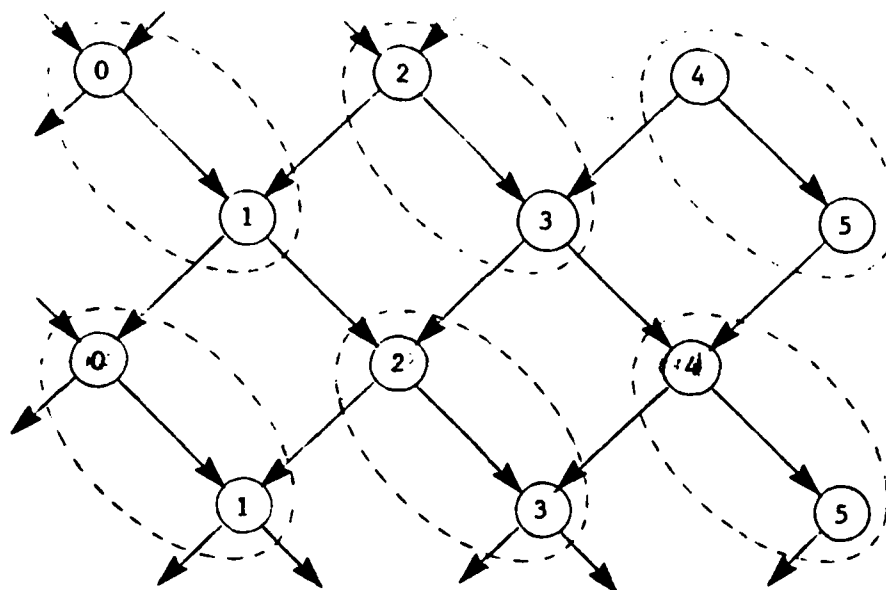


a. Space-Time Diagram

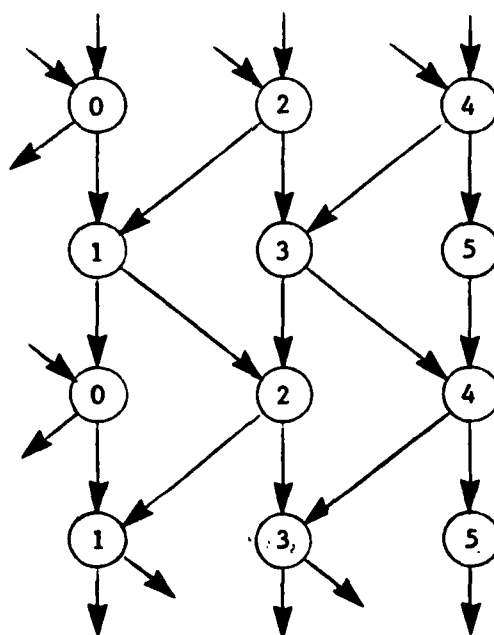


b. Self-Timed Block-Diagram

Figure 4-4. Hardware Implementation of an Iterative MCN.



a. Direct Form



b. Interleaved Form

Figure 4-5. Resource Sharing by Interleaving

4.2 COMPLETELY REGULAR MCNs

A completely regular MCN is one that can be represented by a regular multidimensional grid, and in which all input-output maps associated with the vertices are identical. Thus, the vertices of a completely regular MCN can be mapped into points of the multidimensional grid Z^n in the n -dimensional Euclidean space R^n , where Z denotes the set of integers; the arcs of a completely regular MCN become vectors (n -tuples of real numbers) representing the directed straight lines connecting points of the grid Z^n . Clearly, not all points in Z^n correspond to vertices of the MCN. Those that do determine the domain of the MCN in Z^n . The requirement of complete regularity translates into the statement that the vectors (arcs) emanating from any point (vertex) in do not depend upon the choice of vertex. Consequently, the entire MCN is characterized by:

(i) the set of dependence vectors $\{d_i\}$ emanating from a single vertex;

(ii) the domain $\Gamma \subset Z^n$;

and

(iii) the input-output map f

$$f: (x_1, \dots, x_p) \rightarrow (y_1, \dots, y_p)$$

associated with every vertex in the domain Γ .

A curious consequence of this definition is that the input-output map f has the same number of inputs and outputs, since the number of arcs emanating from a point in Γ is always the same as the number of arcs converging to a point.

Not every set of dependence vectors $\{d_i\}$ determines a valid MCN. For instance, the directed graph representing an MCN has to be acyclic. In terms of dependence vectors this means that it is impossible to find positive integers $\{k_i\}$ such that $\sum k_i d_i = 0$. Another requirement is

that the ancestry of every vertex $v \in \Gamma$ (i.e., the set of all points from which v can be reached) has to be finite. This constraint is trivial if Γ is a finite set; however, if Γ is infinite (as is often the case with signal processing algorithms) this constraint implies that Γ has to be bounded in the directions $\{-d_i\}$.

In the sequel we shall focus upon completely regular MCNs in Z^3 , because such MCNs correspond to space-time representation of planar systolic-array-like architectures (see [23] - [31]). We shall impose the constraint of causality resulting from the association of 'time' with one of the coordinate axes in Z^3 and examine the flow of data through the architecture in terms of the dependence vectors characterizing the MCN.

4.2.1 Space-Time Representations in Z^3

MCNs in Z^3 are characterized by 3-dimensional dependence vectors $\{d_i\}$, which we shall represent by row vectors of length 3. The collection of all dependence vectors

$$D := [d_i]_{i=1}^P \quad (4.1)$$

forms a $p \times n$ matrix, which we shall call the dependence matrix. The boundary of the domain Γ can always be described as a polyhedron. It will be sufficient for our purposes to consider only convex polyhedra, and in fact, only those that can be characterized in terms of the dependence vectors (see Section 5.4 for a further discussion of this choice).

The interpretation of MCNs in Z^3 as space-time representations of hardware architectures imposes the additional constraint of causality: every dependence vector must have a positive time coordinate, since computation and propagation of data cannot be accomplished in zero time. Moreover, since data cannot propagate faster than the speed of electromagnetic waves in metallic conductors, the directions of dependence vectors must lie within a certain cone, the time-like cone in the space-time continuum. By appropriate scaling of space and time coordinates we can reduce this condition to the requirement

$$d_1 [0 \ 0 \ 1]^T \geq 1 \quad (4.2)$$

which means that the third coordinate of d_1 must be (an integer) larger or equal to 1.

The association of time with the third coordinate of dependence vectors allows us to express the finite ancestry condition in simple form. The exclusion of ancestors that are infinitely remote from a given vertex in the domain Γ is equivalent to the requirement that Γ be a subspace of the positive half space of Z^3 , i.e., the half space corresponding to non-negative time coordinates. Moreover, since hardware must always be finite, the spatial extent of Γ must be bounded. Thus, the only direction in which Γ may remain unbounded is that of positive time, corresponding to a computation that continues indefinitely in time (e.g., a filtering of an infinite time-series), but produces results (outputs) at regular intervals.

Vertices in Γ that share the same spatial coordinates are considered as representing the same hardware processor at different instances in time. Regularity implies that such isospatial vertices are spread in time at regular intervals. This interval, which is the same for all processors, will be called the periodicity index of the architecture. The periodicity index corresponding to a given dependence matrix D is the smallest solution π of the equation

$$\eta D = \pi [0 \ 0 \ 1] \quad (4.3)$$

where η is any row vector with integer (possibly negative) entries. To prove this result we notice that ηD is an integer combination of dependence vectors; moreover, if $v(x_1, y_1, t_1)$ and $v(x_2, y_2, t_2)$ are two distinct vertices in Γ , then the vector connecting these vertices can always be expressed in the form ηD for an appropriate (possibly nonunique) row vector η . If the two vertices share the same spatial coordinates, then their interconnecting vector is colinear with $[0 \ 0 \ 1]$, and so (4.3) is satisfied for some η, π . Finally, the smallest temporal displacement is obtained when π is minimized in (4.3). The periodicity index π can, actually, be evaluated without an exhaustive search through all possible integer vectors of η that satisfy (4.3), as is demonstrated in Section 4.2.2.

The most important attribute of the space-time representation of a completely regular MCN is the invariance of the MCN under coordinate transformations in space-time. This is so because coordinate transformations do not affect the interconnection pattern of the space-time representation, and consequently leave the corresponding directed graph unaltered. In the case of regular space-time representations it is sufficient to consider the effect of linear coordinate transformations; this is done in detail in Sections 5 and 6.

4.2.2 Spatial Projection of MCNs in Z^3

The first two coordinates in a three-dimensional space-time can be interpreted as physical space. When a space-time representation is projected into the plane formed by the first two coordinates, vertices represent computing agents (i.e., processors) and arcs represent physical interconnections (i.e., wires). The projection amounts to the truncation of each dependence vector to its first two coordinates, viz.,

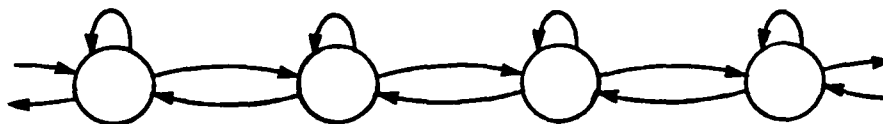
$$D_s := D \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad (4.4)$$

The truncated dependence matrix D_s ('s' stands for 'spatial') is usually sufficient to characterize the architecture, since we commonly assume that each dependence vector represents a computation that requires a unit of time, and consequently

$$D = D_s \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (4.5)$$

This assumption is violated only when D has a periodicity index $\pi(D) > 1$ and, in addition, D contains a dependence vector of the form $[0 \ 0 \ \tau]$. This dependence vector is truncated to $[0 \ 0]$, so τ cannot be recovered unless $\tau = \pi$ or $\tau = 1$. These, in fact, are the only two possible values for τ as explained in Section 6.4.

The truncated dependence matrix can be pictorially represented by a conventional block-diagram such as Figure 4-5. Each truncated dependence vector is represented by an arc with the appropriate spatial displacement, while truncated dependence vectors of the form $[0 \ 0]$, which correspond to local memory, are represented by self-arcs.



a. Block-Diagram Representation

$$D = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad D_s = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix}$$

b. Dependence and Truncated Dependence Matrices

Figure 4-5. Example of a Regular Hardware Architecture

The truncated equivalent of (4.3) becomes

$$\eta D_s = 0 \quad (4.6)$$

so that every feasible choice of η corresponds to an undirected loop in the 2-dimensional block-diagram representation. Thus, every feasible η is obtained by considering all possible loops in the block-diagram representation. If there are no self-loops on vertices, then D_s contains no zero row and (4.5) holds. Consequently, by (4.3),

$$\eta D[0 \ 0 \ 1]^* = \eta[1 \ 1 \ . \ . \ 1]^* = \pi$$

so π is obtained by adding up the entries of η . This is, in fact, done by counting each arc along the loop as 1 if it coincides with the orientation of the loop and as -1 if it points in the reverse direction. Since the smallest value of π is required, only the shortest loops need to be considered. We shall show in Section 5.3 that π never exceeds 3 and is seldom larger than 1.

4.3 MODULAR DECOMPOSITION OF MCN MODELS

The conversion of a given algorithm into an MCN model is based upon the assumption that the fundamental building blocks--the processors--are implementable in hardware. This is indeed so if each processor represents a few scalar operations, which can be handled by any contemporary computing agent. However, signal processing algorithms are rarely specified in this convenient form; most often they are represented by block-diagrams whose blocks involve multivariable manipulations, and in particular, matrix algebra. The construction of a computer program or a hardware implementation for such MCNs requires to decompose each multichannel processor into a subnetwork of scalar processors. One way to achieve this decomposition is by storing standard subnetworks for commonly used operations such as matrix addition, multiplication and inversion in a library and invoke this information whenever the need arises. However, the experience with signal processing schemes indicates that better results are

obtained by matching the method of decomposition to the structure of the problem. By a judicious application of the principle of modular decomposition [32] we obtain completely regular MCNs which are perfectly suited for implementation in VLSI, and have a much higher throughput than those obtained by mapping matrix operations directly into parallel hardware.

There is, at present, no simple test to establish the applicability of the principle of modular decomposition to a given multichannel operation. When the operation is linear in all its operands the necessary and sufficient conditions for modular decomposability can be stated in simple terms as described below.

4.3.1 Modular Decomposition of Linear Multivariable Filters

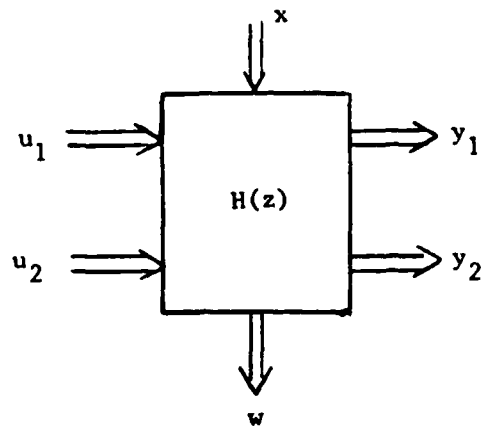
Let $H(z)$ be the transfer function of a multivariable filter with an equal number of inputs and outputs. Suppose that the inputs have been aggregated into three multichannel inputs x, u_1, u_2 and the outputs have been respectively aggregated into w, y_1, y_2 (Figure 4-6). Thus,

$$\begin{bmatrix} y_1(z) \\ y_2(z) \\ w(z) \end{bmatrix} = H(z) \begin{bmatrix} x(z) \\ u_1(z) \\ u_2(z) \end{bmatrix} \quad (4.7)$$

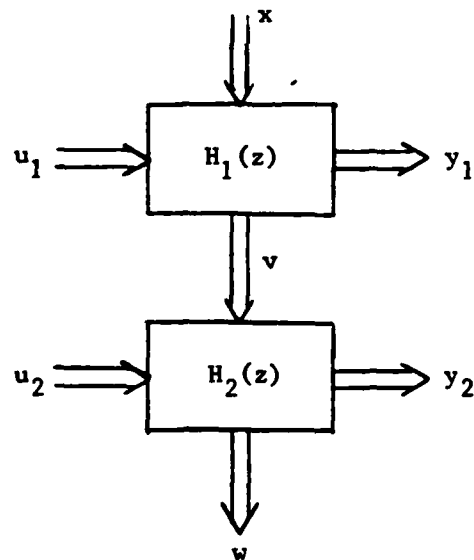
Suppose that there exist transfer functions $H_1(z), H_2(z)$ such that

$$H(z) = \begin{bmatrix} I_p & 0 \\ 0 & H_2(z) \end{bmatrix} \begin{bmatrix} H_1(z) & 0 \\ 0 & I_q \end{bmatrix} \quad (4.8)$$

where p is the number of channels in u_1 and q is the number of channels in u_2 . Then the filter can be, clearly, decomposed in the form described by Figure 4-6. Conversely, if such a decomposition exists, then (4.8) will hold. Thus, the existence of the factorization (4.8) is a necessary and sufficient condition for the decomposability of a multivariable filter as described by Figure 4-6.



a. Before Decomposition



b. After Decomposition

Figure 4-6. Modular Decomposition of a Linear Multivariable Filter

To make the decomposability condition more explicit we shall describe the transfer functions $H_i(z)$ in terms of blocks, viz.,

$$H_i(z) = \begin{bmatrix} A_i(z) & B_i(z) \\ C_i(z) & D_i(z) \end{bmatrix}, \quad i = 1, 2 \quad (4.9)$$

The corresponding decomposition of $H(z)$ yields, via (4.8), (4.9) the identity

$$H(z) := \begin{bmatrix} H_{11}(z) & H_{12}(z) & H_{13}(z) \\ H_{21}(z) & H_{22}(z) & H_{23}(z) \\ H_{31}(z) & H_{32}(z) & H_{33}(z) \end{bmatrix} = \begin{bmatrix} A_1(z) & B_1(z) & 0 \\ A_2(z)C_1(z) & A_2(z)D_1(z) & B_2(z) \\ C_2(z)C_1(z) & C_2(z)D_1(z) & D_2(z) \end{bmatrix} \quad (4.10)$$

This means that some elements of $H_i(z)$ can be uniquely determined from those of the given $H(z)$, for instance

$$A_1(z) = H_{11}(z), \quad D_2(z) = H_{33}(z), \text{ etc.}$$

Since the blocks $B_i(z)$, $C_i(z)$ are square (while A_i , D_i need not be square), we also obtain, assuming nonsingularity of square transfer functions

$$A_2(z) = H_{21}(z)C_1^{-1}(z) \quad (4.11a)$$

which implies

$$H_{21}(z)C_1^{-1}(z)D_1(z) = H_{22}(z) \quad (4.11b)$$

Similarly

$$C_2(z) = H_{31}(z)C_1^{-1}(z) \quad (4.12a)$$

which implies

$$H_{31}(z)C_1^{-1}(z)D_1(z) = H_{32}(z) \quad (4.12b)$$

As a consequence, the elements of $H(z)$ cannot be arbitrary. In fact,

$$\begin{bmatrix} H_{21}(z) & H_{22}(z) \\ H_{31}(z) & H_{32}(z) \end{bmatrix} \begin{bmatrix} C_1^{-1}(z)D_1(z) \\ -I \end{bmatrix} = 0 \quad (4.13a)$$

which means that for all z

$$\text{rank} \begin{bmatrix} H_{21}(z) & H_{22}(z) \\ H_{31}(z) & H_{32}(z) \end{bmatrix} \leq n$$

where n is the number of channels in the signals x, v, w . In fact, if we insist that square transfer functions are, generically, invertible, then $H_{31}(z)$ is an $n \times n$ nonsingular matrix, so that

$$\text{rank} \begin{bmatrix} H_{21}(z) & H_{22}(z) \\ H_{31}(z) & H_{32}(z) \end{bmatrix} = n \quad (4.13b)$$

Conversely, if (4.13b) holds, then

$$C_1^{-1}(z)D_1(z) = H_{31}^{-1}(z)H_{32}(z) \quad (4.14)$$

Choosing $C_1(z)$ arbitrarily we obtain $A_2(z)$, $C_2(z)$, $D_1(z)$ via (4.11a), (4.12a) and (4.14), respectively.

In summary, a necessary and sufficient condition for a decomposition of the form (4.10) to exist is that the rank condition (4.13b) holds and in addition

$$H_{13}(z) = 0 \quad (4.15)$$

Given a transfer function $H(z)$ that satisfies the decomposability conditions we can always compute $H_1(z)$, $H_2(z)$. In fact, we may choose $C_1(z)$ arbitrarily, and the simplest choice is $C_1(z) = I$. This yields the decomposition

$$H(z) = \begin{bmatrix} I_p & 0 & 0 \\ 0 & H_{21}(z) & H_{23}(z) \\ 0 & H_{31}(z) & H_{33}(z) \end{bmatrix} \begin{bmatrix} H_{11}(z) & H_{12}(z) & 0 \\ I_n & H_{31}^{-1}(z)H_{32}(z) & 0 \\ 0 & 0 & I_q \end{bmatrix} \quad (4.16)$$

SECTION 5

CLASSIFICATION OF ARCHITECTURES

Completely regular MCNs were characterized in the previous section in terms of their dependence vectors. It was also indicated that MCNs with different dependence vectors may nevertheless be equivalent, namely they will have equivalent space-time representations. The equivalence of completely regular MCNs is easy to verify, since it amounts to the existence of a nonsingular linear transformation relating the dependence matrices of the MCNs in consideration.

The study of equivalence can be carried out at several different levels of abstraction. At the lowest (most detailed) level each completely regular MCN is represented by a dependence matrix

$$D := [d_i]_{i=1}^P \quad (5.1)$$

where d_i are row vectors of length 3 whose first two coordinates represent the planar space of integrated circuits and the third coordinate represents time. Thus, for instance, the MCN of Figure 5-1 is characterized by the dependence matrix

$$D = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Notice that the time coordinate of all three dependence vectors equals to 1, reflecting the assumption that each dependence vector represents a computation that requires a unit of time. This assumption can, of course, be modified to incorporate computations with unequal processing times. Notice also that the direction of dependence vectors coincides with that of the arrows in Figure 5-1, pointing toward the successors of a given processor, rather than toward the predecessors of the same processor, as in [25].

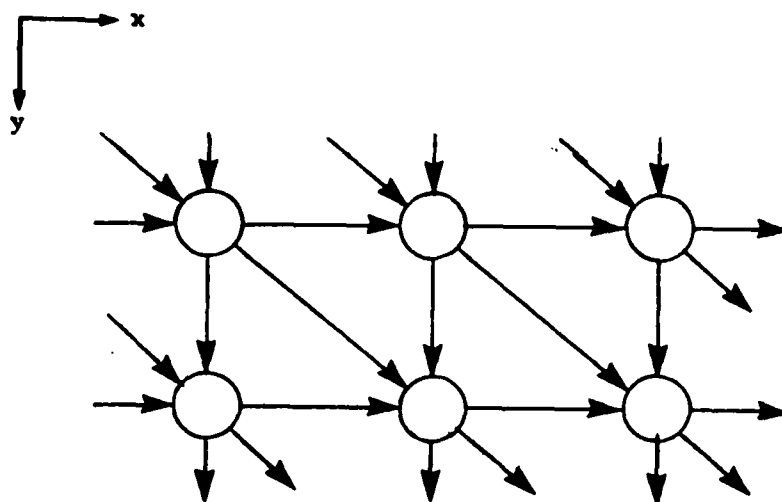


Figure 5-1. Example of a Completely Regular MCN

At the intermediate level of abstraction only the spatial coordinates of each dependence vector are considered. This results in the elimination of the third column of the dependence matrix D , resulting in the truncated dependence matrix D_s

$$D_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

for the example of Figure 5-1. We shall show in the following section that the truncated dependence matrix D_s provides, in fact, a complete, albeit implicit, characterization of the MCN. This characterization can be transformed in a unique manner into the explicit characterization D .

At the highest level of abstraction only the topology of the hardware is considered. This means that the directed graph representing the flow of data is replaced by the corresponding non-directed graph. Thus, for instance, the MCN of Figure 5-1 and that of Figure 5-2 are topologically equivalent, even though the latter has a different dependence matrix, viz.

$$D_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix}$$

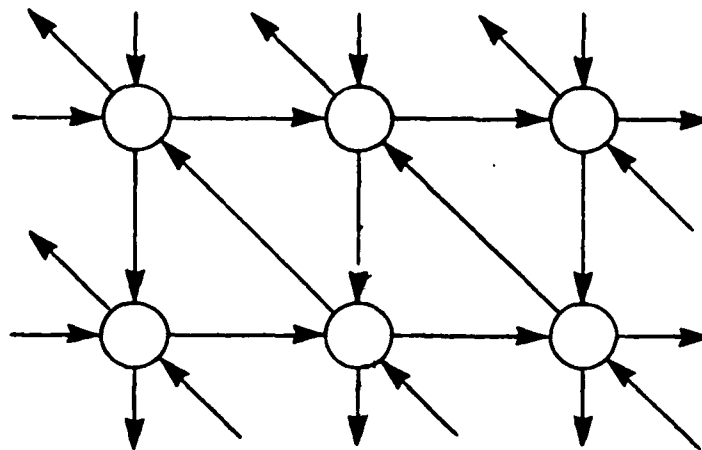


Figure 5-2. A Completely Regular MCN Which is Topologically Equivalent to that of Figure 5-1

This section is devoted to the study of topological equivalence followed by the study of architectural (D_s) equivalence. The more complicated topic of space-time equivalence is presented in the following section, where it is also shown that distinct hardware configurations may, nevertheless, have equivalent space-time representations.

5.1 TOPOLOGICAL EQUIVALENCE

The topic of topological equivalence has been studied by Miranker and Winkler [25], who have shown that there are only three distinct topologies (Figure 5-3):

- (1) The linear topology, with a single dependence vector,

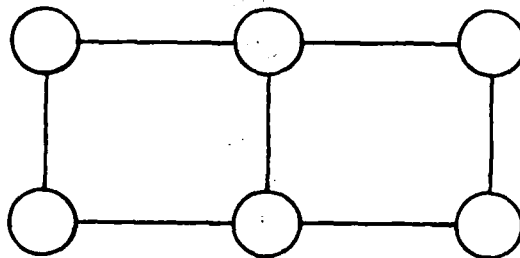
$$D_s = [1 \quad 0]$$

- (2) The rectangular topology, with two dependence vectors,

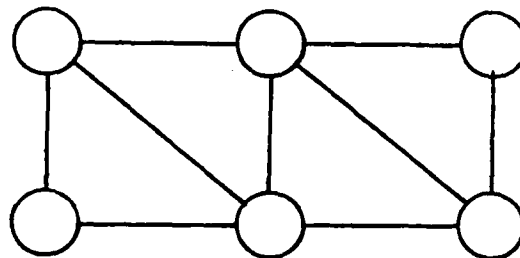
$$D_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



a. The Linear Topology



b. The Rectangular Topology



c. The Hexagonal Topology

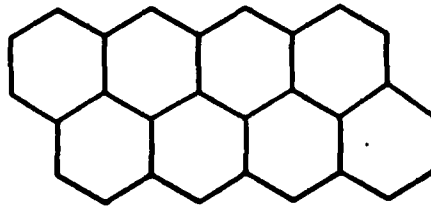
Figure 5-3. The Three Fundamental Topologies

(3) The hexagonal topology, with three dependence vectors,

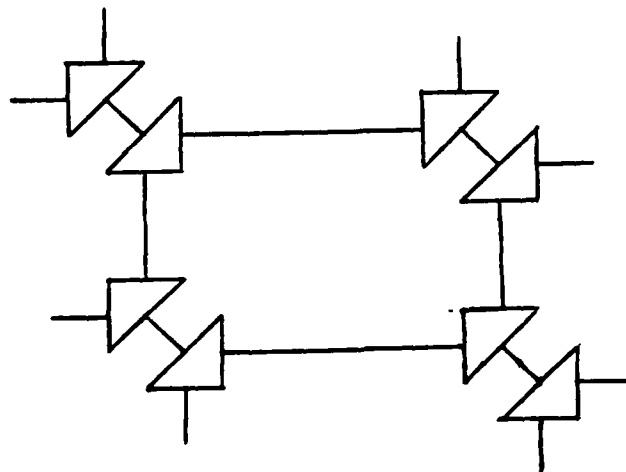
$$D_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Every systolic-array-like architecture can be related by a linear transformation to one of these fundamental topologies. Also, it is impossible to have more than three non-colinear dependence vectors in a planar architecture.

The same conclusion can be reached by a graph-theoretic argument. The graph describing the hardware configuration of a completely regular MCN is clearly a mosaic, i.e., a planar graph in which all faces are bounded the same number of edges and all vertices (except those on the external boundary of the graph) have the same number of incident edges. As is well known, there are only three possible mosaics [21]: triangular, rectangular and hexagonal. The triangular mosaic has vertices of degree 6 and coincides with the hexagonal topology. The rectangular mosaic has vertices of degree 4 and coincides with the rectangular topology. The hexagonal mosaic (Figure 5-4) does not correspond to a completely regular MCN, since it requires two sets of dependence vectors rather than one. However, it can be rearranged by combining pairs of adjacent processors into a single processor (Figure 5-4b), so that the resulting configuration has a rectangular topology. Thus, there are only two mosaics corresponding to completely regular MCNs, which combined with the linear configuration makes a total of 3 fundamental topologies.



a. The Mosaic



b. Rearrangement as a Rectangular Topology

Figure 5-4. The Hexagonal Mosaic

5.2 ARCHITECTURAL EQUIVALENCE

Each of the interconnecting wires in the three fundamental topologies can be associated with two direction vectors, one pointing along the wire in one way, the other in the reverse. This makes a total of three possibilities for each interconnecting wire: (i) $+d$, (ii) $-d$, and (iii) $\pm d$. This means that the linear topology results in $3^1 = 3$ architectures, the rectangular topology in $3^2 = 9$ architectures and the hexagonal topology in $3^3 = 27$ architectures. Since many of these architectures are equivalent, a classification of the distinct architectures is provided in Table 3-1. The nomenclature consists of a capital letter (L, R or H) indicating the topology (linear, rectangular or hexagonal), a digit indicating the number of dependence vectors and a lower case letter, whenever required, to distinguish between architectures which have the same topology and the same number of dependence vectors but are not equivalent, e.g., H3a and H3b. The table lists all equivalent configurations in a single row.

5.3 PERIODICITY ANALYSIS AND THROUGHPUT



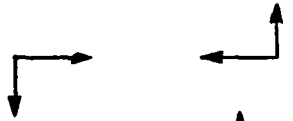

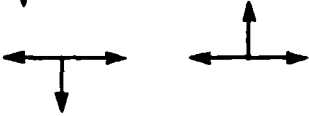
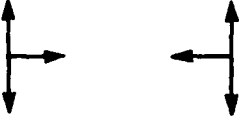
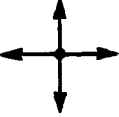

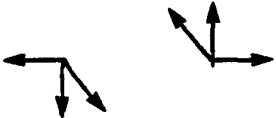




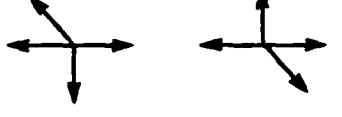


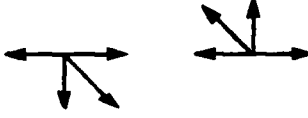
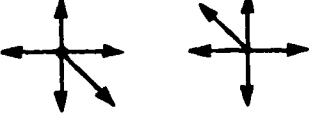
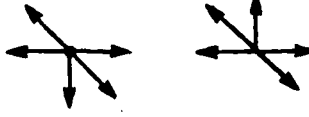

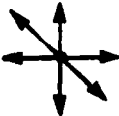
The occurrence of cycles (i.e., closed loops of directed arcs) in the directed graph representing a hardware architecture provides important information about the throughput rate of the architecture. In this subsection we analyze this information and identify the configurations with low throughput.

The periodicity index π of architectures has been defined in Section 4.2. It can be computed either by examining undirected loops in the graph representing the architecture or by solving the equation

$$\eta D_s = 0 \quad (5.2)$$

for every possible row vector η with integer elements, and summing the elements of η . The periodicity index π equals the smallest of these

TABLE 5-1. CLASSIFICATION OF HARDWARE ARCHITECTURES

	Pair 1	Pair 2	Pair 3
L1			
L2			
R2			
R3			
R4			
H3a			
H3b			
H4a			
H4b			
H5			
H6			

sums. If no solution η exists, π is defined to be 1. Following this technique we conclude that L1, R2 have no solution and have a unit periodicity index, while other architectures have solutions, as follows:

- (i) L2 has $\eta = [1 \ 1]$; hence $\pi = 2$.
- (ii) R3 has $\eta = [1 \ 1 \ 0]$; hence $\pi = 2$.
- (iii) R4 has $\eta = [1 \ 1 \ 0 \ 0], [0 \ 0 \ 1 \ 1]$; hence $\pi = 2$.
- (iv) H3a has $\eta = [1 \ 1 \ -1]$; hence $\pi = 1$.
- (v) H3b has $\eta = [1 \ 1 \ 1]$; hence $\pi = 3$.
- (vi) H4a has $\eta = [1 \ 1 \ -1 \ 0], [0 \ 0 \ 1 \ 1], [1 \ 1 \ 0 \ 1]$; hence $\pi = 1$.
- (vii) H4b has $\eta = [1 \ -1 \ 0 \ 1], [0 \ 0 \ 1 \ 1]$; hence $\pi = 1$.
- (viii) H5 has $\eta = [1 \ 0 \ 1 \ 0 \ -1], [1 \ 1 \ 0 \ 0 \ 0], [0 \ 0 \ 1 \ 1 \ 0]$; hence $\pi = 1$.
- (ix) H6 has $\eta = [1 \ 0 \ 1 \ 0 \ -1 \ 0], [1 \ 1 \ 0 \ 0 \ 0 \ 0], [0 \ 0 \ 1 \ 1 \ 0 \ 0], [0 \ 0 \ 0 \ 0 \ 1 \ 1]$; hence $\pi = 1$.

In the sequel we shall measure the throughputs of architectures relative to the throughput of the linear architecture L1 (a classical pipeline). Since the time interval between two successive applications of input equals the periodicity index, the relative throughput of a given architecture is given by the formula

$$\text{relative throughput} = \frac{1}{\text{periodicity index}} \quad (5.3)$$

Thus, the relative throughput of L2, R3, R4 is $1/2$ and that of H3b is $1/3$.

5.4 BOUNDARY ANALYSIS

No assumption has been made up to this point about the shape of the boundary of a given hardware architecture. However, since the shape of the boundary is changed by linear transformation it has to be taken into consideration in the process of classifying architectures. As an example consider the 6 equivalent configurations denoted by H3a (Table 5-1). The truncated dependence matrices of the first and third of these configurations are related by a linear transformation, viz.

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$$

Now assume that the first configuration has a rectangular boundary, which can be characterized by boundary matrix

$$B_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

consisting of all dependence vectors colinear with the boundary. The linear transformation maps this boundary into

$$B'_s = B_s \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$$

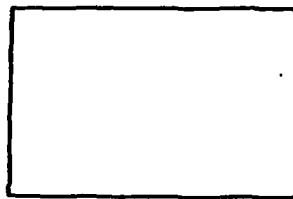
which characterizes a parallelogram rather than a rectangle. Thus, the first H3a configuration with a rectangular boundary is equivalent to the third H3a configuration with a parallelogram boundary. It is not equivalent, however, to the third H3a configuration with a rectangular boundary. Clearly, we need to reclassify the entries of Table 5-1 according to both the dependence matrix and the boundary.

We shall be concerned only with boundaries that satisfy the two following conditions:

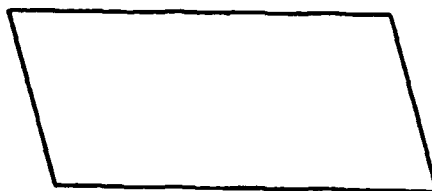
- (1) The boundary curve is a closed convex polygon

- (ii) Each segment of the boundary curve is colinear with some dependence vector.

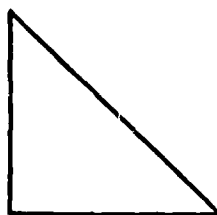
Thus, the only possible directions for the segments of the boundary curve are $[1\ 0]$, $[0\ 1]$ and $[1\ 1]$. Consequently, there are four possible boundary curves (Figure 5-5): rectangle, parallelogram, lower triangle, upper triangle. Of these, only the rectangle-shape boundary can be applied to the linear (1) and rectangular (R) architectures. On the other hand, all four possible boundaries can be combined with hexagonal (H) architectures. However, since linear transformations map rectangles into parallelograms and lower triangles into upper ones, we need only consider the combination of each hexagonal entry of Table 5-1 with either a rectangular or a triangular boundary.



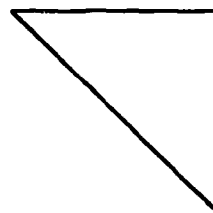
a. Rectangle



b. Parallelogram



c. Lower Triangle



d. Upper Triangle

Figure 5-5. Fundamental Boundary Curves

**TABLE 5-2. CLASSIFICATION OF HARDWARE ARCHITECTURES
WITH RECTANGULAR BOUNDARIES**

L1	L2	R2	R3	R4
1 0	1 0 -1 0	1 0 0 1	1 0 -1 0 0 1	1 0 -1 0 0 1 0 -1

H3 $\alpha\alpha$	H3 $\alpha\beta$	H3b	H4 $\alpha\alpha$	H4 $\alpha\beta$
1 0 0 1 1 1	-1 0 0 1 1 1	1 0 0 1 -1 -1	1 0 0 1 1 1 -1 -1	1 0 0 1 -1 0 -1 -1

H4b α	H4b β	H5 α	H5 β	H6
1 0 0 -1 1 1 -1 -1	1 0 0 1 0 -1 1 1	1 0 -1 0 0 1 0 -1 1 1	1 0 -1 0 0 1 1 1 -1 -1	1 0 -1 0 0 1 1 1 -1 -1

With rectangular boundaries we need to consider matrices of the form

$$\begin{bmatrix} D_s \\ B_s \end{bmatrix} = \begin{bmatrix} D_s \\ I \end{bmatrix} \quad (5.4)$$

Clearly

$$\begin{bmatrix} D_s \\ I \end{bmatrix} \cdot (-I) = \begin{bmatrix} -D_s \\ -I \end{bmatrix} \sim \begin{bmatrix} -D_s \\ I \end{bmatrix}$$

which shows that the reversal of all dependence vectors does not produce a new configuration. The 6 entries in each one of the rows H3a, H4a, H4b, H5 of Table 5-1 can, therefore, be considered as 3 pairs of conjugate configurations. Of these, the second and third pair are still equivalent when combined with rectangular boundaries, but the first pair is different. Thus, the entries of Table 5-1, when combined with rectangular boundaries, can be reclassified as in Table 5-2. This time each architecture is specified by its D_s matrix rather than by a pictorial description as in Table 5-1.

Similarly, we can combine each hexagonal entry of Table 5-1 with a lower triangular boundary. This will again produce two distinct architectures for each one of the rows H3a, H4a, H4b, H5. However, there is no need to do it explicitly, since the resulting configurations can always be obtained by 'cutting' the appropriate hexagonal topology combined with a rectangular boundary along the main diagonal. Thus, it will be sufficient to focus in the sequel upon rectangular boundaries alone.

5.5 SUMMARY

Systolic-array-like architectures have been classified by topology, interconnection pattern and shape of boundary. We have shown that there are only 15 distinct (non-equivalent) architectures (see Table 5-2). We have also shown that it is sufficient to consider only rectangular boundaries which are of practical importance in the process of VLSI layout.

A genealogical chart (Figure 5-6) shows which architectures are contained in any given architecture. In particular, it shows that H6 is the 'universal architecture' for systolic arrays, containing every possible architecture with a smaller number of dependence vectors.

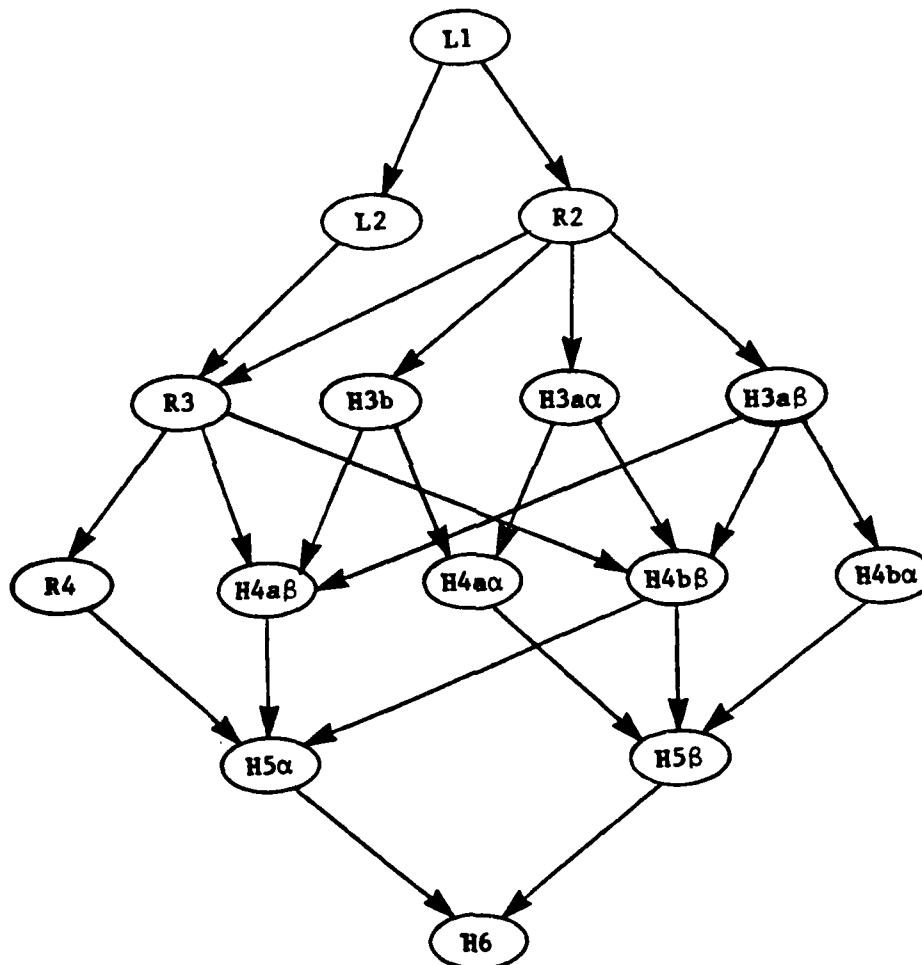


Figure 5-6. Genealogical Chart for Architectures

AD-A146 030

ANALYSIS AND DESIGN METHODOLOGY FOR VLSI COMPUTING
NETWORKS(U) INTEGRATED SYSTEMS INC PALO ALTO CA
H LEV-ARI AUG 84 ISI-46 N00014-83-C-0377

2/2

UNCLASSIFIED

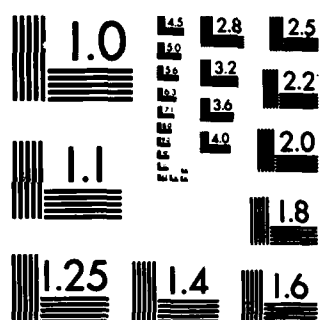
F/G 9/2

NL

END

FORM D

ONE



SECTION 6

CLASSIFICATION OF SPACE-TIME REPRESENTATIONS

The space-time representation of a completely regular MCN was characterized in the previous section by the dependence matrix D . The hardware configuration was obtained by focusing upon the spatial coordinates of the dependence vectors, which resulted in the truncated dependence matrix D_s . It was observed that the temporal coordinate of all the architectures described in Section 5 was always equal to 1, viz.,

$$D = \begin{bmatrix} & 1 \\ D_s & \vdots \\ & 1 \end{bmatrix} \quad (6.1)$$

so that the dependence matrix D can be easily reconstructed for any given D_s via (6.1). The properties of the corresponding space-time diagram can then be deduced by analysis of the dependence matrix D .

6.1 THE FUNDAMENTAL SPACE-TIME CONFIGURATIONS

Each of the fundamental 11 architectures of Table 5-2 determines a fundamental space-time configuration. We shall focus our attention upon the dependence matrix alone, without considering, for the present, the shape of the boundary surface. Thus, equivalence between the fundamental space-time configurations is established by relating the corresponding dependence matrices by linear transformations. A simple analysis (see Appendix F) shows that every dependence matrix with 2 vectors can be transformed into the equivalent (canonical) form

$$D^{(2)} := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

and every dependence matrix with 3 vectors can be transformed into the equivalent form

$$D^{(3)} := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Consequently, $L2 \sim R2$ and $R3 \sim H3a \sim H3b$ where the tilde (\sim) denotes equivalence. For dependence matrices with more than 3 vectors it is convenient to establish first a (nonunique) canonical equivalent, i.e., an equivalent dependence matrix whose first three rows are the identity matrix, viz.,

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{0}{X} & -\frac{1}{X} \end{bmatrix}$$

Some canonical-form equivalents are listed in Table 6-1. The full list of canonical equivalents will be discussed in later sections in conjunction with the specification of boundary surfaces in the three-dimensional space-time continuum.

6.2 ARCHITECTURES WITH LOCAL MEMORY

The preceding analysis was based upon the assumption that processors transmit the results of computations to their immediate neighbors and never store them for further use. However, many applications do involve such storage; this is true, in particular, for adaptive system/parameter identification algorithms that store the identified parameters in fixed location within the array and use the signals that flow through each processor to time-update the locally stored parameters. In this section we consider the architectures obtained by providing each processor with a local memory.

TABLE 6-1. CANONICAL FORM EQUIVALENTS FOR FUNDAMENTAL ARCHITECTURES

L1	L2, R2	R3, H3a, H3b	R4
1 0 0	1 0 0 0 1 0	1 0 0 0 1 0 0 0 1	1 0 0 0 1 0 0 0 1 1 -1 1

H4a	H4b	H5	H6
1 0 0 0 1 0 0 0 1 3/2 -1 1/2	1 0 0 0 1 0 0 0 1 -1/2 1 1/2	1 0 0 0 1 0 0 0 1 -1/2 1 1/2	1 0 0 0 1 0 0 0 1 -1/2 1 1/2 3/2 -1 1/2

Topologically, local memory means the addition of a self-loop to each processor (Figure 6-1). The direction of each interconnecting link can still be chosen in 3 distinct ways, as explained in Section 5.2, resulting in 11 new architectures (Table 6-2). Two important observations have to be made regarding this table:

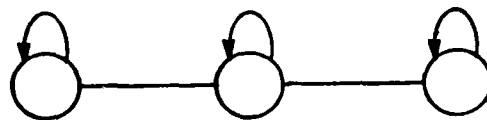
- (i) The number of dependence vectors is larger by one than the number of interconnections. Thus, for instance, RM3 has 4 direction vectors, not 3.
- (ii) The length of the last dependence vector, corresponding to the local memory, equals the temporal displacement between two consecutive occurrences of the same processor in the space-time configuration. Thus, in general, this displacement is 1, except for L2, R3, R4 whose temporal displacement is 2 (corresponding to a periodicity index of 2), and except for H3b whose temporal displacement is 3 (corresponding to a periodicity index of 3).

Local memory can also be used to interleave computations and achieve increased throughput with architectures whose relative throughput without memory is less than 1. This possibility will be discussed in Section 6.4.

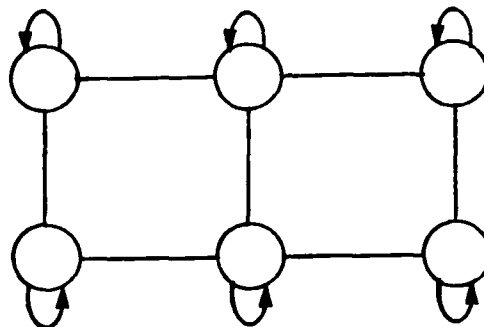
Analysis of equivalence between space-time configurations with local memory reveals that:

- (i) LM1, which has 2 linearly independent dependence vectors, is equivalent to L2, R2.
- (ii) RM2, which has 3 linearly independent dependence vectors is equivalent to R3, H3a, H3b.
- (iii) HM3a, which has 4 dependence vectors, is equivalent to R4.

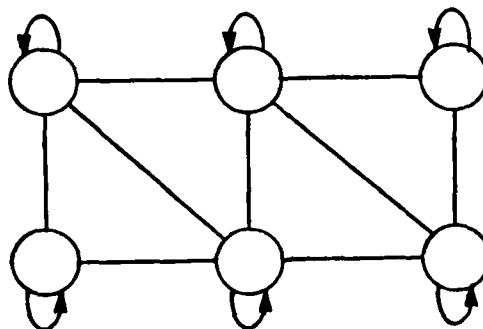
In all three cases we can trade interconnecting links for memory, thereby



a. The Linear Topology with Memory (LM)



b. The Rectangular Topology with Memory (RM)



c. The Hexagonal Topology with Memory (HM)

Figure 6-1. The Three Fundamental Topologies with Local Memory

TABLE 6-2. THE FUNDAMENTAL HARDWARE ARCHITECTURES WITH LOCAL MEMORY

LM1	LM2	RM2	RM3	RM4
1 0 1	1 0 1	1 0 1	1 0 1	1 0 1
0 0 1	-1 0 1	0 1 1	-1 0 1	-1 0 1
	0 0 2	0 0 1	0 1 1	0 1 1
			0 0 2	0 -1 -1
				0 0 2

HM3a	HM3b	HM4a	HM4b	HM5	HM6
1 0 1	1 0 1	1 0 1	1 0 1	1 0 1	1 0 1
0 1 1	0 1 1	0 1 1	0 -1 1	-1 0 1	-1 0 1
1 1 1	-1 -1 1	1 1 1	1 1 1	0 1 1	0 1 1
0 0 1	0 0 3	-1 -1 1	-1 -1 1	0 -1 1	0 -1 1
		0 0 1	0 0 1	1 1 1	1 1 1
				0 0 1	-1 -1 1
					0 0 1

reducing the number of physical wires required to construct a realization of the architecture and simplifying the layout problem for VLSI implementation. Thus, for instance, the R2 architecture which requires a planar network of processors with 4 interconnecting ports at each processor can be replaced by LM1 which requires a linear network of processors with 2 interconnecting ports at each processor and a local memory. Even more remarkably, the same replacement also trades low throughput configurations for high throughput ones.

6.3 BOUNDARY ANALYSIS

The relation between boundary shapes and equivalence between (planar) architectures has been examined in Section 5.4. The combination of topology and boundary has produced 15 distinct architectures which were summarized in Table 5-2. Since each one of these architectures has a rectangular boundary, the resulting space-time configuration always occupies a rectangular prism (with the exception of low-dimensional architectures such as L1, L2, R2 whose space-time configurations occupy 1 or 2-dimensional subspaces).

Since linear transformations change the shape of the boundary, the equivalence between space-time configuration, discussed in Sections 6.1 - 6.2, has to be reexamined to include the effects of boundary transformations. It will be sufficient to carry out this analysis only for collections of space-time configurations which have been declared as equivalent in the preceding sections.

6.3.1 The Configurations LM1, L2, R2

The configurations LM1, R2 can be considered equivalent only when we assume that a single set of inputs is applied to R2 (rather than a time-series). In this case R2 is characterized by

$$\begin{bmatrix} D \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

while LM1 is characterized by

$$\begin{bmatrix} D \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

and the two are equivalent, being related by a linear transformation, viz.,

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

On the other hand, LM1 and R2 are not equivalent to L2 for which

$$\begin{bmatrix} D \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

The D-part of this characterization can be related to the D-part of LM1, viz.,

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ * & * & * \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

where the asterisks denote entries which can be chosen arbitrarily (subject to the nonsingularity constraint of the linear transformation). However, when the dependence matrix is combined with the boundary matrix we obtain

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -1 \\ 1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

which does not match the B-part of L2. When the inverse of this transformation is applied to the dependence and boundary matrices of L2, viz.,

$$\begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -1 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -0 & -0 & -1 \\ -1 & 0 & 1 \\ 1/2 & 0 & 1 \end{bmatrix}$$

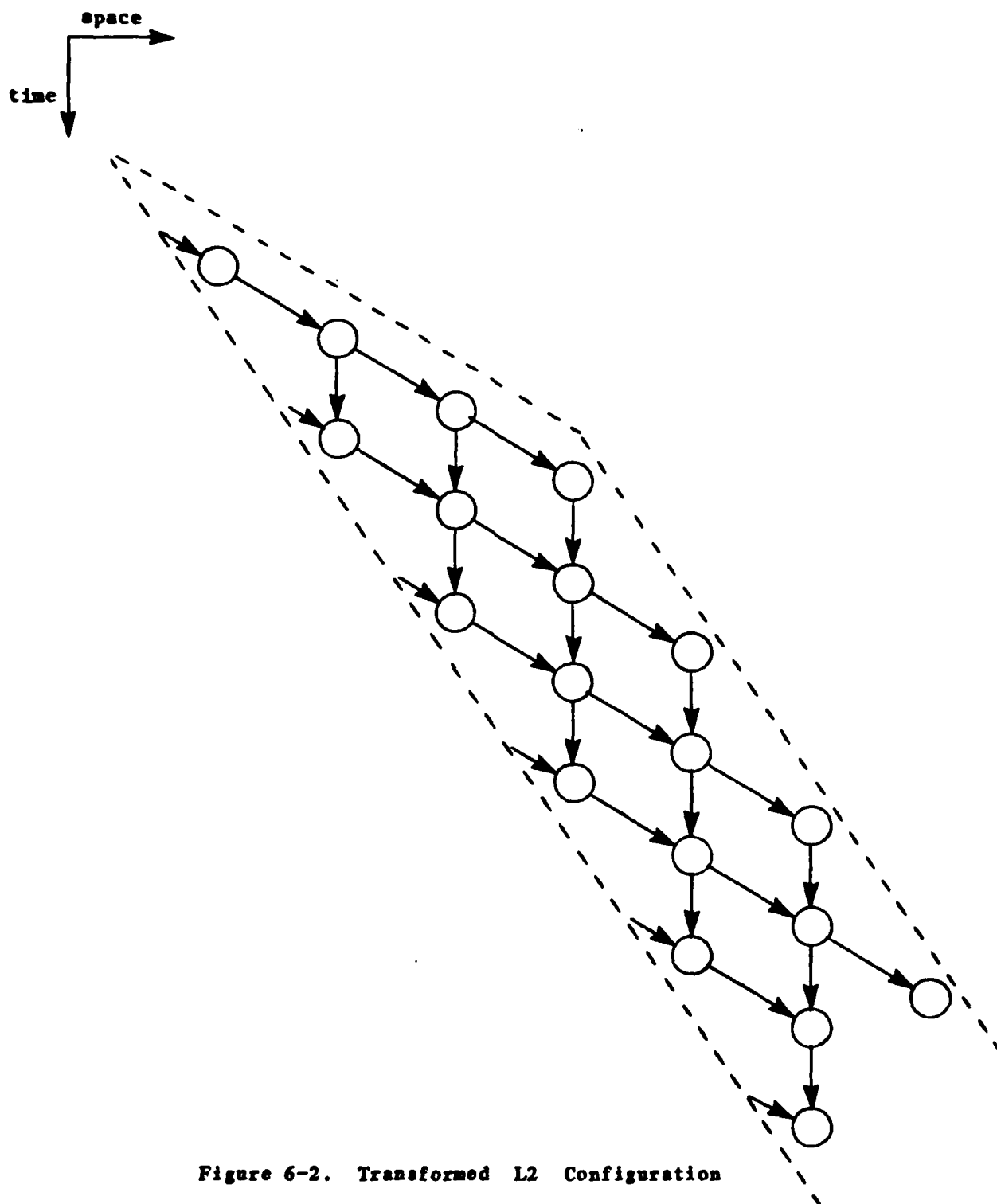


Figure 6-2. Transformed L2 Configuration

the resulting configuration (Figure 6-2) is equivalent to an LM1 configuration of infinite order; the finite active part of the architecture is shifted one cell to the right every time a new input is applied. Thus, in summary, LM1 and R2 are equivalent to each other but not to L2.

6.3.2 The Configurations RM2, R3, H3a, H3b

The truncated boundary matrix of these configurations was chosen in Section 5.4 as

$$B_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

namely, the rectangular boundary. The corresponding boundary surface in the space-time continuum is, therefore, characterized by the boundary matrix

$$B = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.1)$$

When this boundary matrix is combined with the dependence matrices of RM2, R3, H3aa, H3aβ, H3b, equivalence is destroyed. For instance, trying to relate H3aa to RM2 we obtain

$$\begin{bmatrix} D \\ B \end{bmatrix}_{H3aa} \cdot \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ -1 & 1 & -1 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ -1 & 0 & -1 \\ -1 & 0 & -1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The resulting D-part coincides with the dependence matrix of RM2, but the boundary surface is different. The configuration obtained above by transforming H3aa is in fact an RM2 hardware of infinite order in which a finite active segment shifts along the diagonal by one cell each time a set of inputs is applied to the array. This is, in fact, precisely what happens in systolic arrays for matrix multiplication. The configuration H3aa (of Weiser and Davis [4]) is suited for multiplying banded matrices. When the same problem is implemented on an RM2 configuration (of S.Y. Kung [7]) most cells in the array are idle while a small active rectangle,

corresponding to the bandwidth of the given matrices, shifts along the main diagonal of the array. In analogy, while multiplying two matrices with no structure is carried out efficiently by an RM2 array, solving the same problem on an H3a configuration involves many idle cells and a small active segment that shifts along the main diagonal.

6.3.3 The Configurations HM3a, R4

These configurations have the same boundary matrix, given by (6.1), as RM2, R3, H3a and H3b. Since their dependence matrices are different, we conclude that HM3a, HM3a β , R4 are distinct configurations when the shape of boundary surface is taken into account.

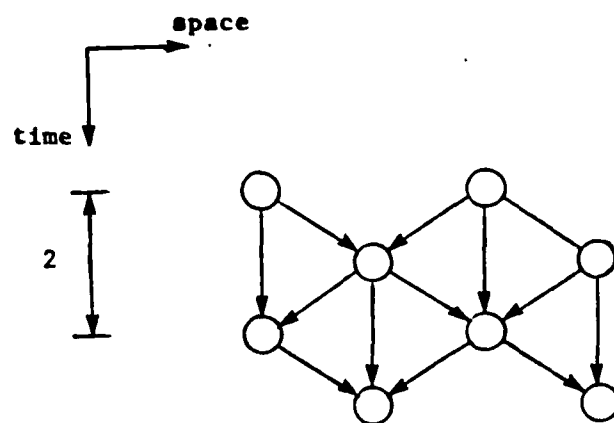
6.3.4 Summary

When boundary considerations are taken into account each of the 15 architectures of Table 5-2 is distinct and cannot be related by equivalence to any other architecture in this table. Incorporating local memory results in doubling the total number of distinct configurations to 30.

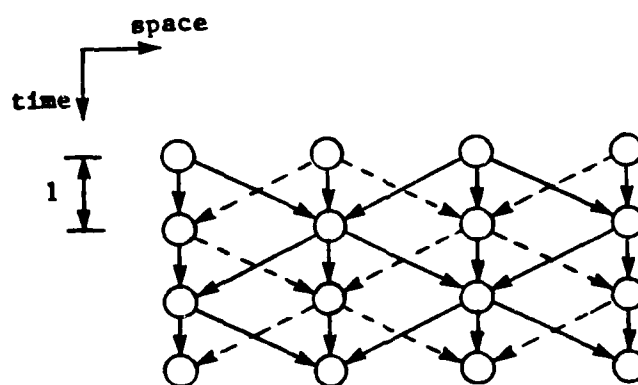
6.4 INTERLEAVING ARCHITECTURES BY LOCAL MEMORY

The introduction of local memory in Section 6.2 involved the assumption that locally stored data remain in memory until required, which makes particular sense in data driven realization. Consequently, the duration of storage for some architectures (L2, R3, R4, H3b) was longer than one time unit. This fact can be used to construct new architectures with higher throughput, by interleaving computations in time and connecting the interleaved computations via the local memory.

The simplest example of such construction is the architecture L2. Without interleaving the throughput of L2, LM2 is $1/2$ (Figure 6-3a). With interleaving, which involves superimposing in time two L2 schemes and interconnecting them via local memory, the resulting LIM2 configuration



a. Without Interleaving (LM2)



b. With Interleaving (LM2)

Figure 6-3. Interleaving via Local Memory

(Figure 6-3b) has throughput = 1. A similar approach produces the architectures RiM3, RiM4 and HiM3b, whose characterizations are given in Table 6-3. The difference between these architectures and their noninterleaved counterparts is the shortening of the local memory dependence vector from either [0 0 2] or [0 0 3] to [0 0 1].

TABLE 6-3. DEPENDENCE MATRICES FOR INTERLEAVED ARCHITECTURES

LiM2	RiM3	RiM4	HiM3b
1 0 1	1 0 1	1 0 1	1 0 1
-1 0 1	-1 0 1	-1 0 1	0 1 1
0 0 1	0 1 1	0 1 1	-1 -1 1
	0 0 1	0 -1 1	0 0 1
		0 0 1	

6.5 SUMMARY

Space-time configurations have been classified by topology, interconnection pattern, shape of boundary, existence of local memory and interleaving. The 15 fundamental architectures of Table 5-2 give rise to another 15 configurations involving local memory. These, in turn, give rise to 4 interleaved configurations, producing a total of 34 distinct space-time configurations.

Ignoring the shape of the boundary surface results in 20 distinct configurations:

- | | |
|---|--|
| 1) L1 | 11) H4 $\alpha\alpha$, H4 $\alpha\beta$ |
| 2) LM1, L2, R2 | 12) H4 $\beta\alpha$, H4 $\beta\beta$ |
| 3) LM2 | 13) RM4 |
| 4) LiM2 | 14) RiM4 |
| 5) RM2, R3, H3 $\alpha\alpha$, H3 $\alpha\beta$, H3 β | 15) HM4 $\alpha\alpha$, HM4 $\alpha\beta$ |
| 6) RM3 | 16) HM4 $\beta\alpha$, HM4 $\beta\beta$ |
| 7) RiM3 | 17) H5 α , H5 β |
| 8) HM3 $\alpha\alpha$, HM3 $\alpha\beta$, R4 | 18) HM5 α , HM5 β |
| 9) HM3 β | 19) H6 |
| 10) HiM3 β | 20) HM6 |

Ignoring, in addition, the details of local memory (and, consequently, of interleaving) results in 8 distinct configurations only as in Table 6-1.

Choosing the optimal configuration for a given computational scheme requires a specification of both the interconnection pattern and the boundary shape. This can be accomplished only when specific details of the corresponding computational scheme are taken into account (e.g., bandedness of matrices to be multiplied). When only partial information is considered the designer is often able to choose the interconnection pattern but not the boundary. Thus, multiplication of two matrices can be implemented in any of the five equivalent hardware configurations RM2 [7], R3 [10], H3 $\alpha\alpha$ [4], H3 $\alpha\beta$, H3 β [5]. However, RM2 will be optimal if both matrices have no particular structure; R3 will be optimal if only one of the matrices is banded; and H3 $\alpha\alpha$ (or H3 $\alpha\beta$) will be optimal if both matrices are banded. It is an historical curiosity that the first systolic array for matrix multiplication, H3 β , is never optimal, because it has relative throughput of 1/3 and is otherwise equivalent to H3 α .

SECTION 7

CONCLUSIONS

A modeling and analysis methodology for parallel algorithms and architectures has been presented. Modular computing networks (MCNs) were introduced as a unifying concept that can be used to describe both algorithms and architectures. The representation of an MCN exhibits all the relevant information that characterizes a parallel processing algorithm, from precedence relations and order of execution, through scheduling and pipelinability consideration, to map compositions and global characterization. It clearly displays the hierarchical structure of a parallel system and the multiplicity of choices for hardware implementation. Our methodology applies both to arbitrary (irregular) networks and to iterative ones. Regularity of networks translates directly into regularity of the model we use to describe them and, consequently, into regularity of the associated architectures. Problems of non-executability (deadlocks, safeness, etc.) are insignificant in our methodology and can be easily detected and resolved.

Infinite MCNs, which occur in most signal processing applications, have been characterized. It has been shown that the key property for executability of such networks is structural finiteness (in addition to absence of cycles, of course). Infinite MCNs are most frequently iterative, in which case they are guaranteed to be structurally finite and can be represented by a finite single-layer diagram.

Dimensionality, pipelinability and throughput have been introduced as fundamental structural attributes of MCN models. Throughput computations (see, e.g., [9]) have been established as a direct consequence of the notion of schedule, which applies to every MCN model. The wavefront concept [4,7,8] has been shown to be a natural outcome of associating schedules with iterative networks. Systolic-array-like architectures were modeled and analyzed via the concept of completely regular MCNs.

A classification of canonical realizations for completely regular modular computing networks has been presented. Three levels of abstraction were considered: topology, architecture and space-time representation. The analysis revealed 3 canonical topologies, 15 canonical architectures and 34 canonical space-time configurations. It was shown that the unique canonical counterpart of any given topology, architecture or space-time configuration is obtained via a simple (and unique) transformation of the corresponding dependence and boundary matrices. It was also shown that only rectangular boundaries are required to implement any canonical realization. While ignoring boundary details allows some flexibility of design, it also results in inefficient implementations, as explained in Section 6.5.

It is interesting to observe that only a small fraction of the architectures described in this memo have actually been used in the design of parallel algorithms. The most commonly encountered architectures are the linear ones (L2, L1M) which are used for linear filtering (= convolution, polynomial multiplication) and related computations. Next comes the rectangular architecture RM2 and its equivalents--R3, H3a, H3b--which are used in matrix products, matrix triangularizations, solutions of linear equations, QR-factorizations for eigenvalue problems, and adaptive multichannel least-squares algorithms. Thus, all applications involved, to date, only architectures with 3 dependence vectors or less. Notice also that the classical pipeline (L1) has no use as a signal processing architecture.

SECTION 8
TECHNICAL PUBLICATIONS

The following technical papers have been written under contract number NO0014-83-C-0377.

1. H. Lev-Ari, 'Modular Computing Networks: A New Methodology for Analysis and Design of Parallel Algorithms/Architectures,' ISI Technical Memo, ISI-29.
2. S.Y. Kung, 'On Supercomputing with Systolic/Wavefront Array Processors,' Special Issue on Supercomputing, IEEE Proc., July 1984.
3. H. Lev-Ari, 'Canonical Realizations of Completely Regular Modular Computing Networks,' ISI Technical Memo, ISI-41.
4. H. Lev-Ari, 'A New Methodology for Representation and Analysis of Parallel Algorithms and Architectures,' in preparation.

REFERENCES

- [1] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981.
- [2] S.Y. Foo and G. Musgrave, 'Comparison of Graph Models for Parallel Computation and Their Extension,' 1975 International Symposium in CHDLs and Their Applications, pp. 16-21.
- [3] S.L. Johnson and D. Cohen, 'A Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks,' in H.T. Kung, et al. (eds.), VLSI Systems and Computations, Computer Science Press, 1981.
- [4] U. Weiser and A. Davis, 'A Wavefront Notation Tool for VLSI Array Design,' in H.T. Kung, et al., *ibid.*
- [5] H.T. Kung, 'Why Systolic Architectures?', IEEE Computer, pp. 37-46, January 1982.
- [6] M.C. Chen and C.A. Mead, 'Concurrent Algorithms as Space-Time Recursion Equations,' in S.Y. Kung, et al. (eds.), Modern Signal Processing and VLSI, Prentice Hall, 1984.
- [7] S.Y. Kung, et al., 'Wavefront Array Processor: Language, Architecture and Applications,' IEEE Trans. Comp., Vol. C-31, pp. 1054-1066, Nov. 1982.
- [8] S.Y. Kung, 'VLSI Array Processors for Signal Processing,' Proceedings of Arab Summer School in Modern Signal Processing, Aug. 1983.
- [9] H.V. Jagadish, T. Kailath, J.A. Newkirk, and R.G. Mathews, 'Pipelining in Systolic Arrays,' submitted to the Seventeenth Asilomar Conference on Circuits and Systems, Pacific Grove, 1983.
- [10] S.K. Rao and T. Kailath, 'VLSI and the Digital Filtering Problem,' submitted to MIT Conference on Advanced Research in VLSI, 1983.
- [11] A.A. Markov, 'The Theory of Algorithms,' Amer. Math. Soc. Trans.
- [12] F. Hennie, Introduction to Computability, Ch. 1, Addison-Wesley, 1977.
- [13] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, Data Structures and Algorithms, Ch. 1, Addison-Wesley.
- [14] D.F. Robinson and L.R. Foulds, Digraphs: Theory and Techniques, Gordon and Breach, 1980.

- [15] C. Berge, The Theory of Graphs and Its Applications, Methuen, London, 1964.
- [16] C.L. Seitz, 'System Timing,' in C.A. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.
- [17] Y. Malachi and S.S. Owicki, 'Temporal Specifications of Self-Timed Systems,' in H.T. Kung, et al., *ibid.*
- [18] J.B. Dennis, 'Data-Flow Supercomputers,' IEEE Computer, pp. 48-56, Nov. 1980.
- [19] W.W. Wadge, 'An Extensional Treatment of Dataflow Deadlock.'
- [20] F. Commoner, A.W. Holt, S. Even, and A. Pnueli, 'Marked Directed Graphs,' J. Comp. Syst. Sci., Vol. 5, pp. 511-523, 1971.
- [21] O. Ore, Graphs and Their Uses, Math. Assoc. of America, Yale, 1963.
- [22] P. LaGuernic, A. Beneveniste, and T. Gautier, 'Signal: Un Langage pour le Traitement du Signal,' IRISA Research Report No. 206, May 1983.
- [23] J.P. Roth and L.S. Levy, 'Equivalence of Hardware and Software,' Research Report RC 9464, IBM Watson Center, Yorktown Heights, NY, May 1982.
- [24] M.C. Chen and C.A. Mead, 'Concurrent Algorithms as Space-Time Recursion Equations,' in S.Y. Kung, et al. (eds.), Modern Signal Processing and VLSI, Prentice Hall, 1984.
- [25] W.L. Miranker and A. Winkler, 'Spacetime Representations of Systolic Computational Structures,' IBM Research Report RC 9775, Dec. 1982.
- [26] P.R. Cappello and K. Steiglitz, 'Unifying VLSI Array Design with Linear Transformations in Space-Time,' Technical Report TRCS 83-03, University of California, Santa Barbara, Dec. 1983.
- [27] D.I. Moldovan, 'On the Design of Algorithms for VLSI Systolic Arrays,' Proceedings of the IEEE, Vol. 71, pp. 113-120, Jan. 1983.
- [28] P. Quinton, 'The Systematic Design of Systolic Arrays,' IRISA Report No. 193, France, Apr. 1983.
- [29] H. Lev-Ari, 'Modular Computing Networks: A New Methodology for Analysis and Design of Parallel Algorithms/Architectures,' ISI Technical Memo, ISI-29, Dec. 1983.
- [30] B. Lisper, 'Description and Synthesis of Systolic Arrays,' The Royal Institute of Technology Report TRITA-NA-8318, Stockholm, Sweden, 1983.
- [31] C.J. Kuo, B.C. Levy, B.R. Musieus, 'The Specification and Verification of Systolic Wave Algorithms,' MIT Report LIDS-P-1368, Cambridge, MA, March 1984.

- [32] H. Lev-Ari, 'Modular Architectures for Adaptive Multichannel Lattice Algorithms,' Proc. 1983 IEEE, Int. Conf. on ASSP, pp. 455-458, Apr. 1983.

APPENDIX A
PROOF OF THEOREM 2.2 FOR INFINITE MCNs

If the MCN has an execution then it must be acyclic, as was pointed out at the beginning of Section 2.3. To prove the converse we shall construct an execution for an arbitrary acyclic, structurally-finite MCN.

First, notice that, by Theorem 2.1, the inputs of the MCN can be numbered. Let us, therefore, denote the inputs by $\{z_i; 0 \leq i < \infty\}$. Next, recursively define a sequence of sets of variables $\{M_i\}$ according to the following rule:

$$M_0 := \{z_0\}$$

$$M_{i+1} := \{A_0(M_i), z_{i+1}\}$$

Thus, each set contains one new inputs of the MCN and all the immediate successors of the preceding set. The sets M_i are clearly disjoint, and, in view of the local-finiteness property, each M_i set is finite. Moreover, every variable of the MCN is included in some M_i set, because every variable is either a global input or a finite successor of some global input. Thus, the cascade

$$M_0 * M_1 * M_2 * \dots$$

is, in fact, a representation of the network as a cascade of finite (disjoint) subnetworks. Each M_i set is finite, hence has an execution with a finite number of levels. If we replace each M_i by its execution, we obtain a refinement of the previous representation, viz.,

$$s_{00} * s_{01} * \dots * s_{10} * s_{11} * \dots * \dots$$

PREVIOUS PAGE
IS BLANK

where $\{S_{ij}\}$ are the levels corresponding to the set M_1 . Since each S_{ij} is finite, this is clearly an execution of the global MCN.

APPENDIX B
ADMISSIBLE ARCHITECTURES

A composition of processors is called admissible if the following three conditions are satisfied:

- (i) There are no dangling inputs or outputs.
- (ii) There are no directed cycles.
- (iii) The architecture is structurally finite.

Each of the processors comprising an architecture can itself be a composition of more elementary processors. The hierarchical nature of the admissibility property implies the following result.

Theorem B.1

An admissible composition of admissible architectures is itself an admissible architecture.

Proof:

The theorem states that the three properties making up admissibility should be exhibited by the composite architecture, if they were exhibited by each of the subnetworks.

- (i) The composite architecture has no dangling terminals, because every terminal is connected to some subnetwork (by admissibility of the composition) and no subnetwork has dangling terminals (by admissibility of the subnetworks).
- (ii) The composite architecture has no cycles because neither the subnetwork nor the composition has cycles.

(iii) Structural finiteness is made up of the three following properties: Local finiteness, finite ancestries, and countability of connected components. Local finiteness is inherited by the composite architecture because composition does not change the number of inputs/outputs of processors within each subnetwork. To prove that the finite ancestry property is also inherited by the composite architecture it will be sufficient to consider a single variable x . Suppose that x belongs to some subnetwork \mathcal{G}_1 . By the admissibility of the composition, \mathcal{G}_1 has a finite number of ancestor subnetworks. The ancestry of x is obtained by tracing back the ancestry relation through the finite collection of subnetworks $a(\mathcal{G}_1)$. And since each subnetwork is admissible, the portion of $a(x)$ within each ancestor of \mathcal{G}_1 is also finite, hence $a(x)$ itself is finite. Finally, an admissible composition has a countable number of subnetworks (see Theorem 2.1) and each subnetwork has, by assumption, a countable number of connected components. Hence, the total number of connected components in the composite network is countable, too.

APPENDIX C
PROOF OF THEOREM 2.3
MINIMAL EXECUTIONS OF FINITE MCNs

Every execution determines a numbering $E()$ of the variables of an MCN, viz.,

$$x \in S_i \longleftrightarrow E(x) = i$$

This integer valued function satisfies the inequality (see Section 3.1)

$$E(x) - 1 \geq \max_y \{E(y); y \in a(x)\} \quad (C.1)$$

Every finite directed acyclic graph has a unique numbering $\hat{E}()$ of its arcs (or equivalently of its vertices) that satisfies the equality

$$\hat{E}(x) - 1 = \max_y \{\hat{E}(y); y \in a(x)\} \quad (C.2)$$

This well-known result (see, e.g., [14]) implies that every finite executable MCN has a unique execution that satisfies (C.2). We shall call this unique execution minimal for reasons that will become clear in the sequel.

Let $E()$ be an arbitrary non-minimal execution. Then, there exists some variable x for which the strict inequality

$$E(x) - 1 > \max_y \{E(y); y \in a(x)\}$$

holds. This means that x is evaluated several steps after all its ancestors became available. Consequently, the numbering of x can be modified to $1 + \max_y \{E(y); y \in a(x)\}$ without violating the precedence

relation. We shall refer in the sequel to this modification as an elementary shift.

Each execution is a series-parallel combination and consequently has a well defined input-output map. Elementary shifts replace expressions of the form $e^*e^*\dots^*p$ by expressions of the form $p^*e^*\dots^*e$ (see Figure C-1). If the physically justifiable identity

$$p^*e = e^*p \quad (C.3)$$

is added as an axiom of the theory of MCNs (see Appendix D), we conclude that input-output maps remain invariant under elementary shifts. This leads to the following result.

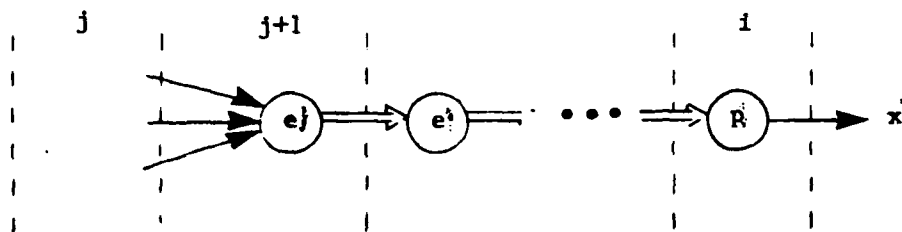
Theorem C.1

Every execution $E()$ of a finite MCN can be transformed by a finite number of elementary shifts into the unique minimal execution.

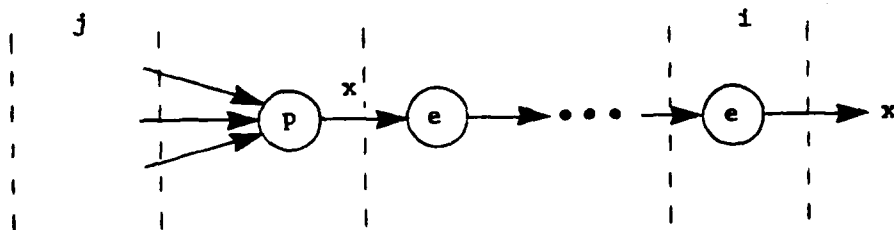
Proof:

The minimal execution $E()$ is constructed by the following simple algorithm (see, e.g., [14]):

- (i) Put all the global inputs of the MCN in \hat{S}_0 .
- (ii) For $i = 0, 1, 2, \dots$ put all the immediate successors of members of \hat{S}_i in \hat{S}_{i+1} .



a. Before the Shift



b. After the Shift

Figure C-1. The Effect of an Elementary Shift.

$$E(x) = i; \max_y \{E(y); y \in a(x)\} = j.$$

Now, if $E()$ is a nonminimal execution we transform it into $\hat{E}()$ by the following rule:

For $i = 0, 1, 2, \dots$ shift all members
of \hat{S}_i from $E(x)$ to $\hat{E}(x) = i$.

Since the MCN is finite, a finite number of shifts will transform $E()$ into $\hat{E}()$. Notice that each variable is shifted exactly once. Also notice that by its construction, the number $\hat{E}(x)$ is equal to the lengths of the shortest path connecting x to some global input. Hence, $\hat{E}(x)$ cannot be further reduced.

Corollary C.1.1

The minimal execution $\hat{E}()$ satisfies $\hat{E}(x) \leq E(x)$ for every variable x and for every execution $E()$.

Corollary C.1.2

A finite executable MCN has a unique well-defined input-output map. This is so because all executions define the same map, by Theorem C.1.

Proof of Theorem 2.3

Corollary C.1.2 establishes the theorem for finite MCNs. For infinite networks it will be sufficient to prove that for every execution $E()$ and for every variable x the map from global inputs to x is unique and does not depend upon the choice of execution. However, $E()$ induces some execution on the finite MCN corresponding to $a(x)$, the finite ancestry of x . Therefore, the map from inputs to x coincides, for every choice of $E()$, with the unique map determined by the minimal execution on $a(x)$.

It is interesting to notice that an infinite MCN does not have, in general, a minimal execution. The construction procedure described in the proof of Theorem C.1 is still valid, but \hat{S}_i are, in general, infinite and do not determine a valid execution.

APPENDIX D
ELEMENTARY EQUIVALENCE TRANSFORMATIONS

The general theory of MCNs does not involve any specific assumptions about the properties of the processor maps $\{f_p\}$. Consequently, there are only a few equivalence transformations that are still valid in this general framework. Most equivalence transformations used with block-diagrams and signal-flow-graphs involve linearity assumptions and do not hold for general nonlinear maps.

Two basic maps, the identity map e and the split map s can be used in conjunction with any MCN manipulation. The identity map leaves its input variables unchanged, viz.,

$$e(x) = x$$

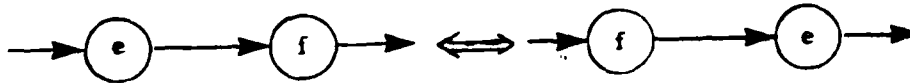
The split map duplicates input variables, viz.,

$$s(x) = (x, x)$$

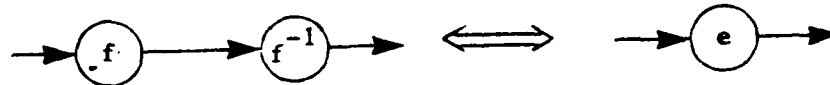
It is possible, of course, to have more than two copies of the same variable, either by introducing a split processor with several outputs, or by using several two-output split processors.

The properties of the identity and split processors give rise to several elementary equivalence transformations (Figure D-1):

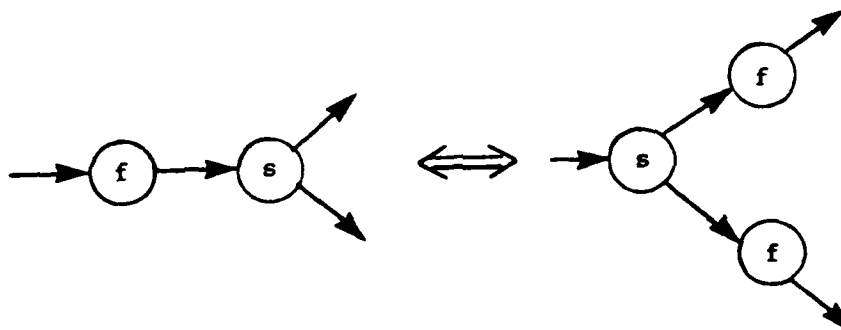
- (a) The identity commutes with any other processor f .
- (b) The cascade of a processor f and its inverse f^{-1} can be replaced by an identity processor, provided the processor f has an inverse.
- (c) The split processor 'commutes' with any processor f .
- (d) Any processor f with two outputs can be replaced by a composition of a split processor and two single output processors f_1, f_2 . The processors f_1, f_2 correspond to the maps from inputs to each of the two outputs, respectively.



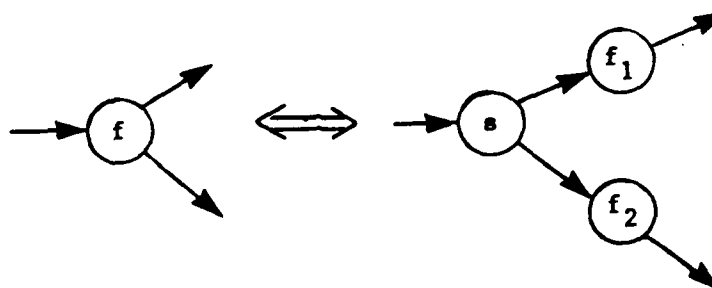
a. Commutativity of the Identity



b. The Inverse Processor



c. Commutativity of the Split



d. Splitting of Multivariable Outputs

Figure D-1. Elementary Equivalence Transformations

APPENDIX E
ANALYSIS OF MATRIX MULTIPLIERS

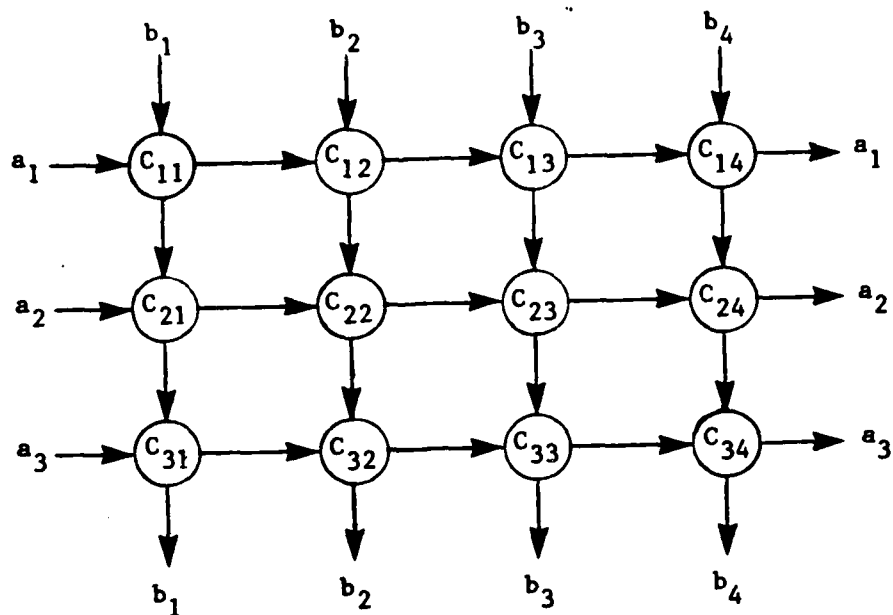
The multiplication of two matrices involves the computation of inner products between every row of one matrix and every column of the other one. To emphasize this interpretation we shall consider in the sequel the product

$$C := A^* B$$

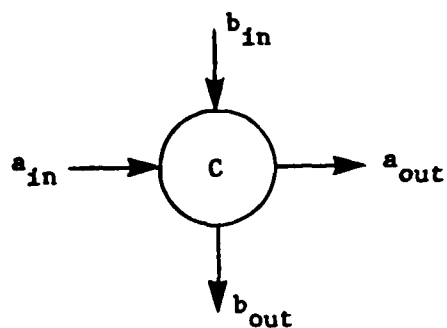
so that the inner products are between columns of A and columns of B . In fact, C_{ij} is precisely the inner product between the i -th column of A and the j -th column of B . Consequently, we can compute the product by feeding the columns of A, B , which we denote by a_i, b_j , into the MCN of Figure E-1. Each input is a column vector which is propagated without modification through the network. The a, b inputs of each processor propagate through without modification and the inner product of the two input vectors is computed inside the processor. This multichannel configuration can be further decomposed by observing that the inner products can be computed recursively, i.e., if $c := a^* b$ where $a = \{a_i\}$, $b = \{\beta_i\}$ are column vectors of length N , then $c = c_N$ where

$$c_i = c_{i-1} + a_i \beta_i, \quad c_0 = 0$$

Thus, every single processor in Figure E-1 is, in fact, a cascade of basic 'multiply and add' processors (Figure E-2). When this decomposition is combined with the architecture of Figure E-1, we obtain the MCN for matrix multiplication. Figure E-3 displays a side view of this 3-D network whose top view is shown in Figure E-1. The complete MCN consists of N horizontal layers such as in Figure E-1 arranged in a vertical stack. Equivalently, we may say the MCN consists of three vertical layers such as in Figure E-3 arranged behind each other. It is important to notice that



a. The Complete Network



$$a_{out} = a_{in}, b_{out} = b_{in}, c = a_{in} \cdot b_{in}$$

b. A Single Processor

Figure E-1. A Basic Matrix Multiplier

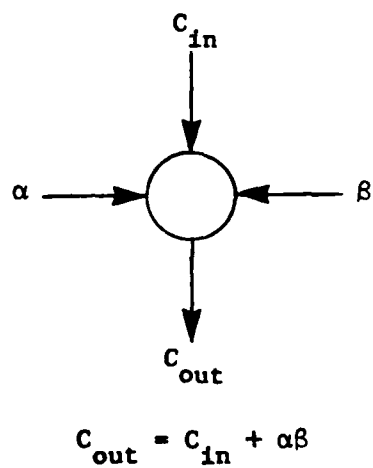
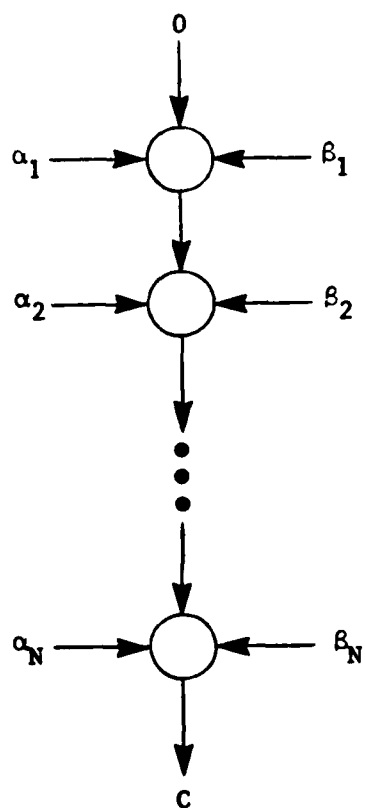


Figure E-2. A Basic Inner Product Array

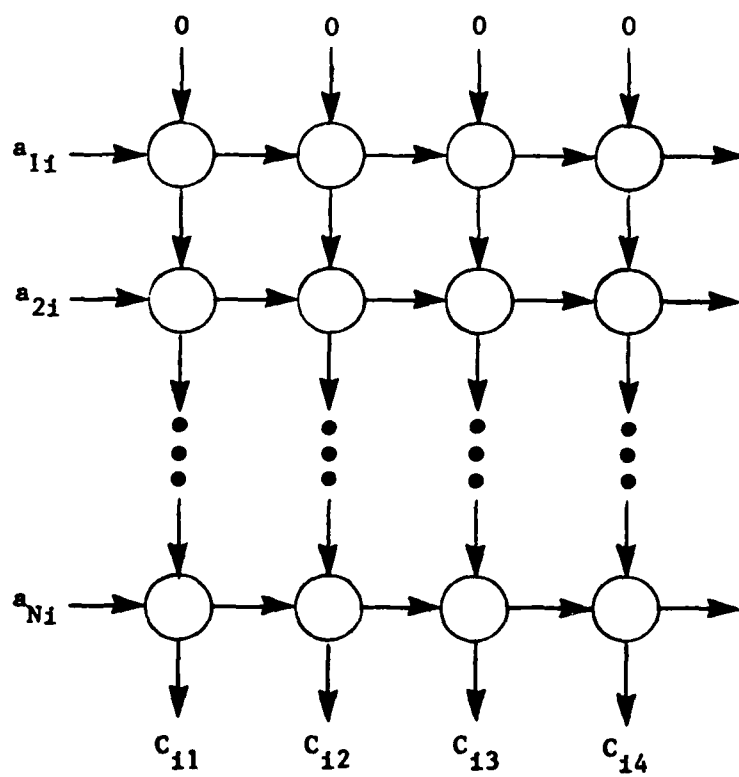


Figure E-3. The MCN of Matrix Multiplication
(side view shows i-th vertical layer)

the direction of the C-paths can be either from top to bottom, as shown in Figure E-3, or from bottom to top. This is a consequence of the commutativity and associativity of addition, viz.,

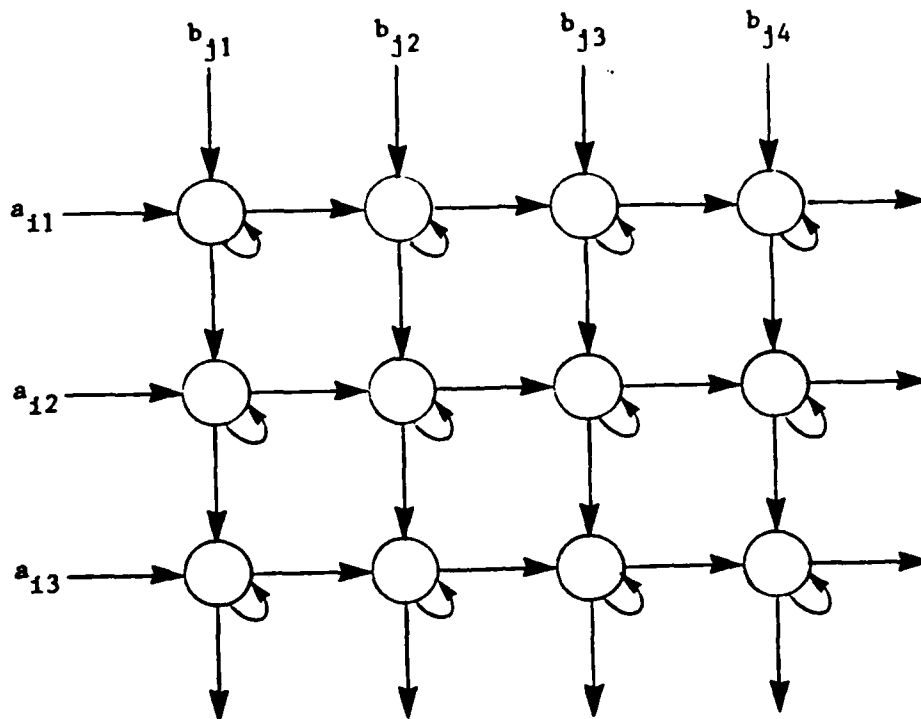
$$\sum_{i=1}^N a_i \beta_i = \sum_{i=N}^1 a_i \beta_i$$

This means that there are two distinct MCNs that correspond to matrix multiplication and they differ only by the direction of the C-paths.

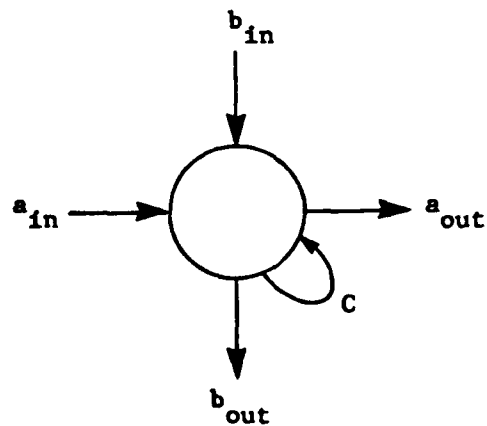
Every architecture for matrix multiplication is equivalent to the MCN of Figure E-3. The various architectures are obtained by imposing additional constraints upon the matrices (i.e., bandedness) and rearranging the resulting reduced MCN as a space-time diagram. The corresponding self-timed block-diagram follows immediately from this rearrangement.

The matrix multiplier of S.Y. Kung [7] is obtained by interpreting the vertical dimension in Figure E-3 as 'time.' Since vertical arrows correspond to local storage, the resulting block-diagram is described in Figure E-4 (notice the similarity with E-1). The elements of each column vector a_i, b_j are fed sequentially into the array and each processor has a self-loop which computes the inner-product $c_{ij} = a_i^* b_j$ recursively in time.

The matrix multiplier of S. Rao [10] is designed for a banded B matrix. It will be sufficient to analyze it for a single column of A, say a_i . The MCN of Figure E-3 now has only one vertical layer, and many processors in this layer have zero inputs and can be eliminated. The resulting reduced MCN is shown in Figure E-5a. Dummy processors, shown in broken line, were added to emphasize the tridiagonal nature of the MCN. A self-timed block-diagram (Figure E-5b) is obtained by considering the diagonal axis as 'time.' It consists of a linear array of identical processors, one for each nonzero diagonal of the banded matrix B. The elements of B are fed into the array by diagonals. The elements of A, C are handled by columns: Every column of A produces a row of C and requires a linear array as in Figure E-5b. It is interesting to notice that the input interval of this matrix multiplier is $\tau_c + \tau_a$ where τ_c is the



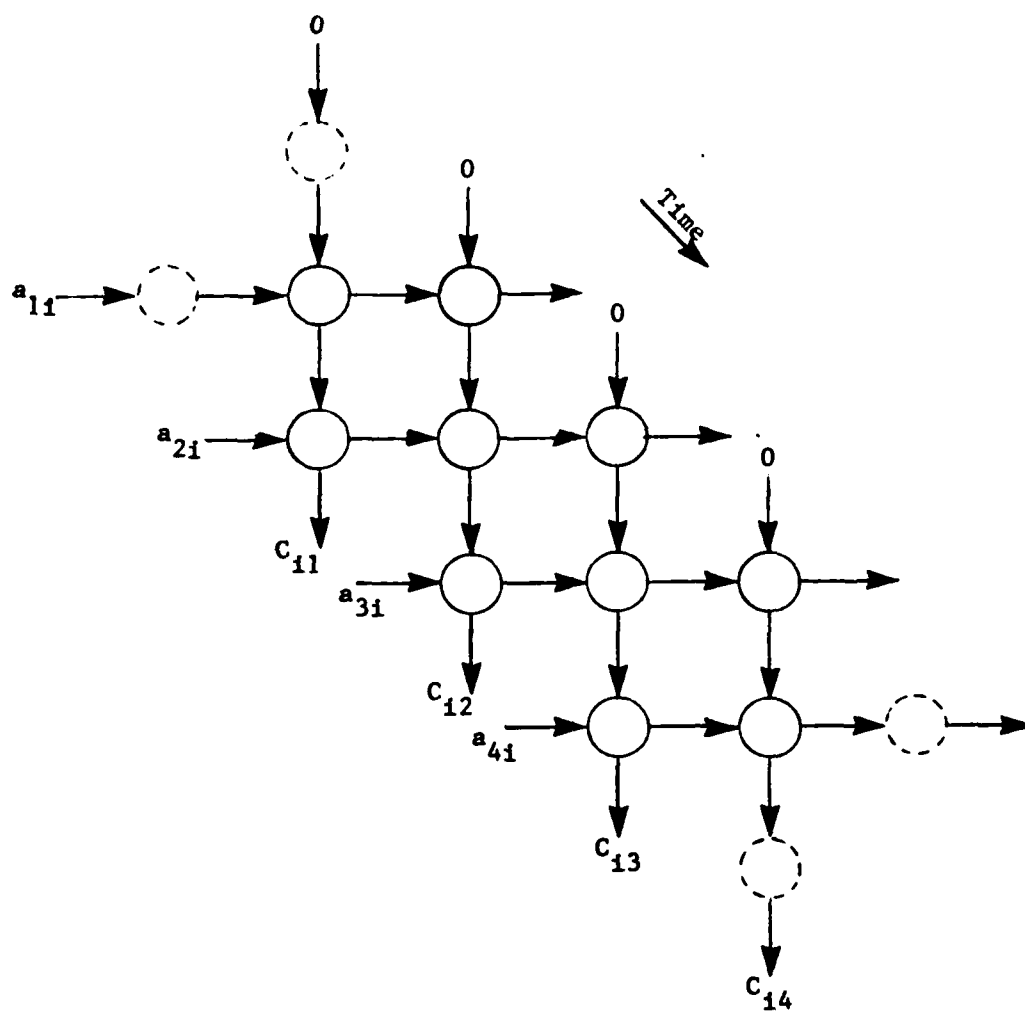
a. Self-Time Block-Diagram



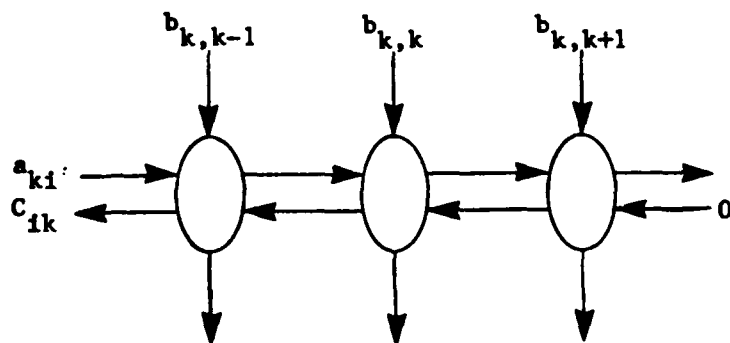
$$a_{out} = a_{in}, b_{out} = b_{in}, c_{new} = c_{stored} + a_{in} b_{in}$$

b. Single Processor

Figure E-4. The Matrix Multiplier of S.Y. Kung



a. Reduced MCN

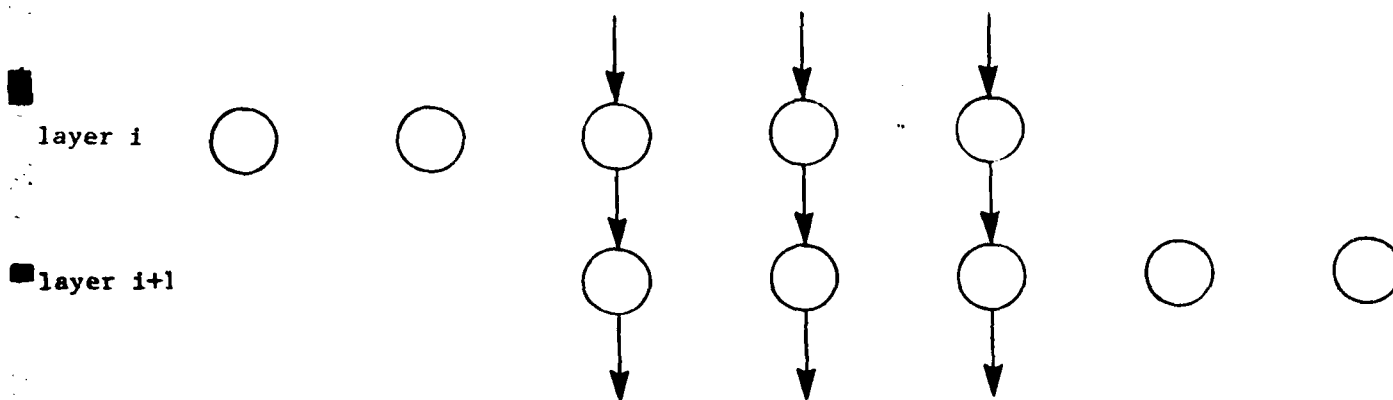


b. Self-Timed Block-Diagram

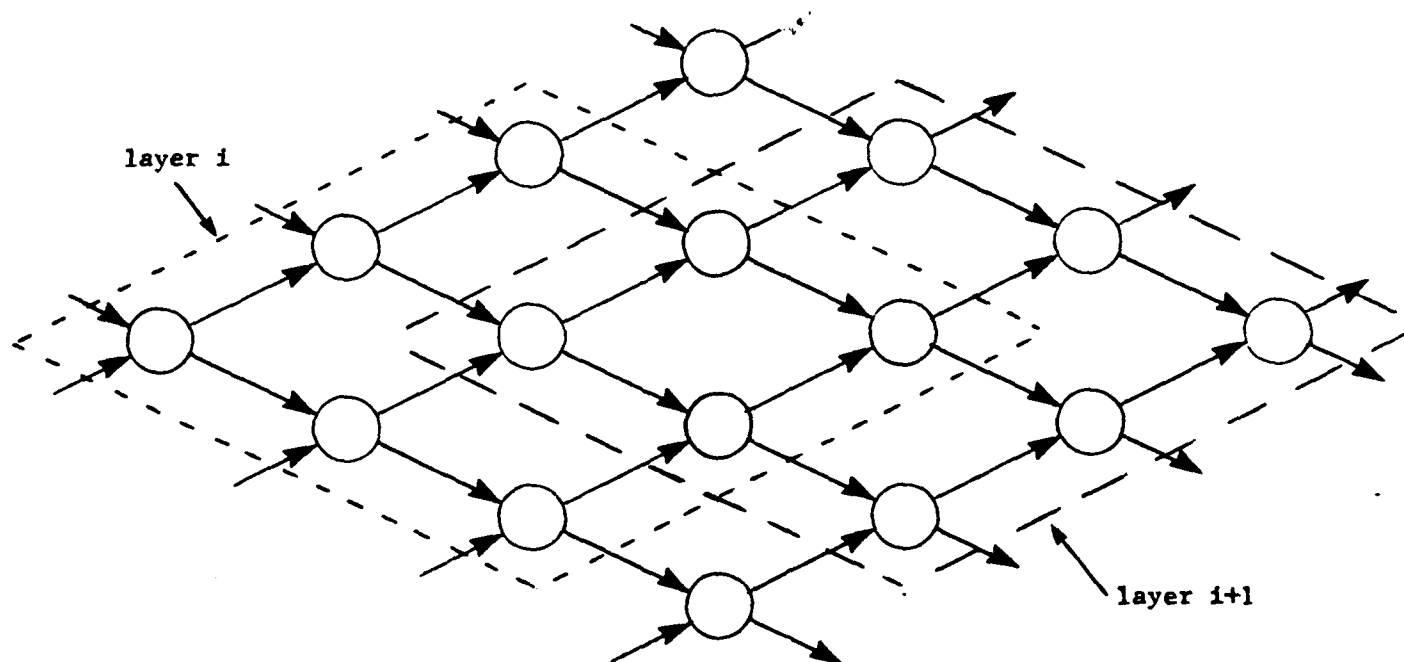
Figure E-5. The Matrix Multiplier of S. K. Rao

time required to compute 'c' and τ_a is the time required to propagate 'a' through one processor. When the direction of the C-path or, equivalently, of the A-path, is reversed the input interval becomes $\tau_c - \tau_a$. Since $\tau_a \ll \tau_c$ the two networks differ only slightly in their throughput. However, we shall presently encounter another example where the reversal of the C-path results in a large increase in throughput.

The matrix multiplier of H.T. Kung is designed for banded A, B matrices. This means that the active processors in the non-reduced MCN of Figures E-1 and E-3 are located within a parallelepiped aligned with one of the main diagonals of the rectangular prism representing the non-reduced MCN. A simple illustration of the reduced MCN is obtained by considering two adjacent horizontal layers (Figure E-6). When we slide the horizontal layers so that they overlap, the resulting network corresponds to H.T. Kung's multiplier (Figure E-7). This network clearly has an input interval of $\tau_c + 2\tau_a$. However, if we reverse the C-path we obtain the configuration of Weiser and Davis [4] (Figure E-8) which has an input interval of $|\tau_c - 2\tau_a|$. The difference between the two multipliers is significant when they are implemented by single rate systolic arrays. In this case $\tau_c = \tau_a = \tau$ so that the former network has an input interval of 3τ while the latter has an input interval of τ !

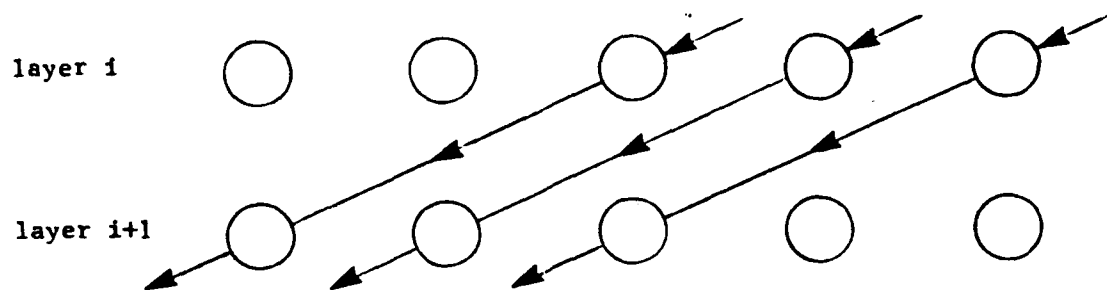


a. Side View

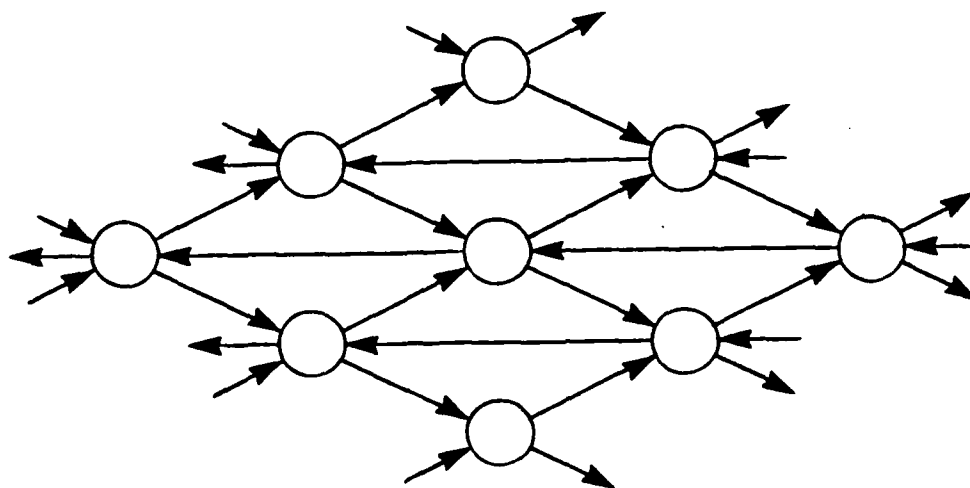


b. Top View

Figure E-6. The Reduced MCN for Banded Matrix Multiplication

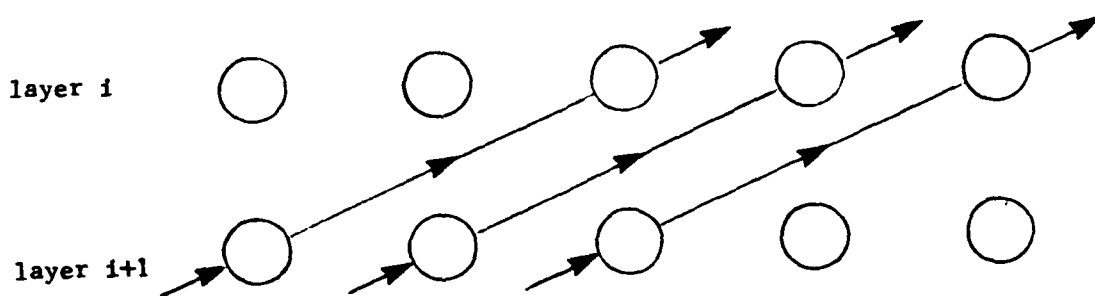


a. Side View

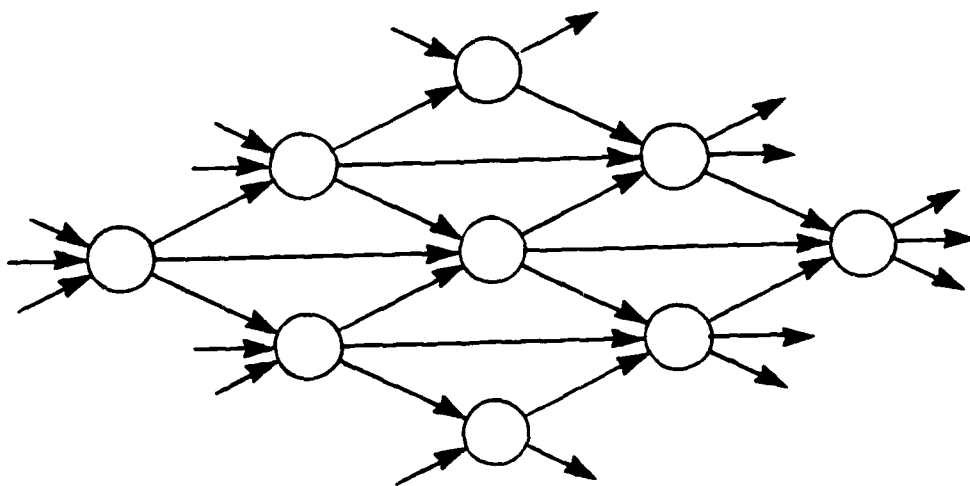


b. Top View

Figure E-7. The Matrix Multiplier of H.T. Kung



a. Side View



b. Top View

Figure E-8. The Matrix Multiplier of Weiser and Davis

APPENDIX F
EQUIVALENCE VIA LINEAR TRANSFORMATIONS

Two dependence matrices, say, D_1, D_2 , are considered equivalent when there exists a nonsingular linear transformation T and a permutation matrix P such that

$$D_2 = PD_1T \quad (F.1)$$

This relation is clearly reflexive (with $P = I, T = I$), symmetric and transitive, so 'equivalence' is indeed an equivalence-type relation.

Denoting the length of dependence vectors by n , and the number of dependence vectors by p , we conclude that every dependence matrix with $p \leq n$ and full (row) rank is equivalent to

$$D^{(p)} := \begin{bmatrix} I_p & 0 \end{bmatrix} \quad (F.2)$$

which will be defined as the canonical equivalent of such dependence matrices. When $p > n$, and the dependence matrix has full (column) rank, we can always find a permutation matrix P so that

$$PD = \begin{bmatrix} I \\ X \end{bmatrix} T \quad (F.3)$$

where T consists of the first n rows of the permuted matrix PD . Thus, the canonical equivalent of dependence matrices with $p > n$ is of the form (F.3) and the properties of D can be studied by examining the structure of the smaller matrix X .

However, since the submatrix X in (F.3) is not unique, it is required first to find all possible canonical equivalents to a given dependence matrix D . This can be done by applying all possible $p!$ permutations P to the rows of D and then computing X via (F.3). However, not all

In summary, once all possible canonical equivalents of a given D have been computed it is relatively easy to test whether some other dependence matrix \hat{D} is equivalent to D . One only needs to compute a single canonical equivalent of \hat{D} and compare it to the collection of canonical equivalents of D : a match indicates that \hat{D} is indeed equivalent to D .

END

FILMED

10-84

DTIC